MARCH 30, 2024

DEEP LEARNING WITH CUDA

LABORATORY 02

ANDRZEJ ŚWIĘTEK
FACULTY OF PHYSICS AND APPLIED INFORMATICS
AGH

Contents

Introduction

In this study, I delve into the critical aspects of model architecture by exploring the effects of varying depth and sizes within neural network structures. Understanding the intricate relationship between model depth, layer sizes, and their impact on performance is paramount in the field of machine learning. Through systematic experimentation and analysis, I aim to uncover insights into how different architectural configurations influence the overall performance of neural networks. By examining a range of architectures and their corresponding performances across various tasks, I seek to elucidate the optimal design principles that govern the construction of robust and efficient models.

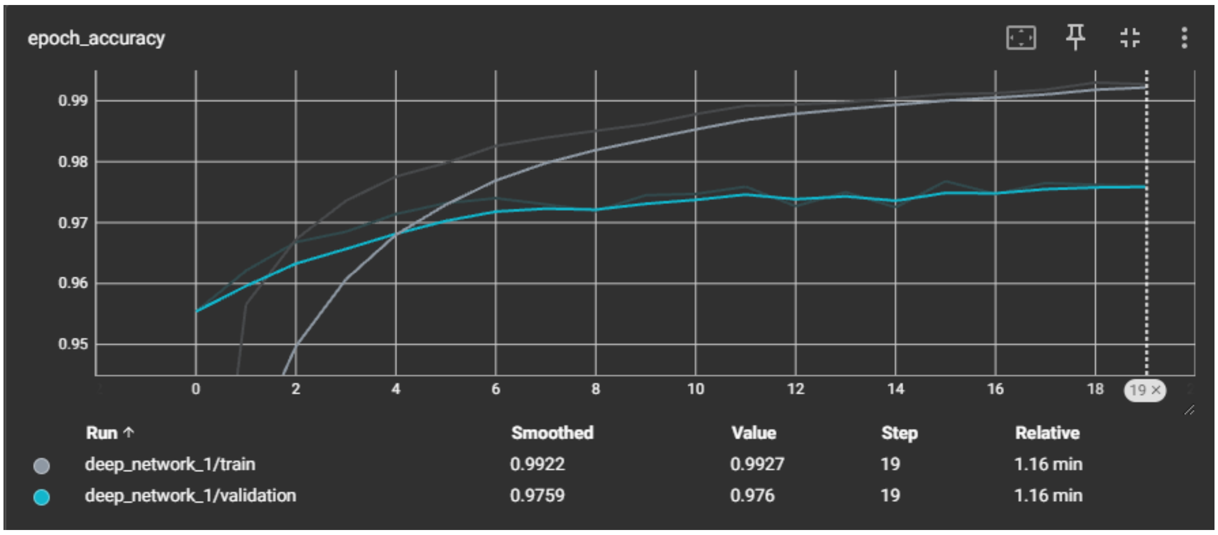
Different Models Architectures

For all models described below I apply a batch normalization after every layer starting from input layer up until output layer (excluded obviously). Additionally, right before the last layer I conducted Dropout on neuron adhering to the principle it’s a good practice to do so for some of last layers. For most of layers I used ReLU activation function – unless emphasized otherwise for specific test-case. When training number of epochs was set to be 20 for all instances and batch size to 128 as the conclusion from previous laboratories.

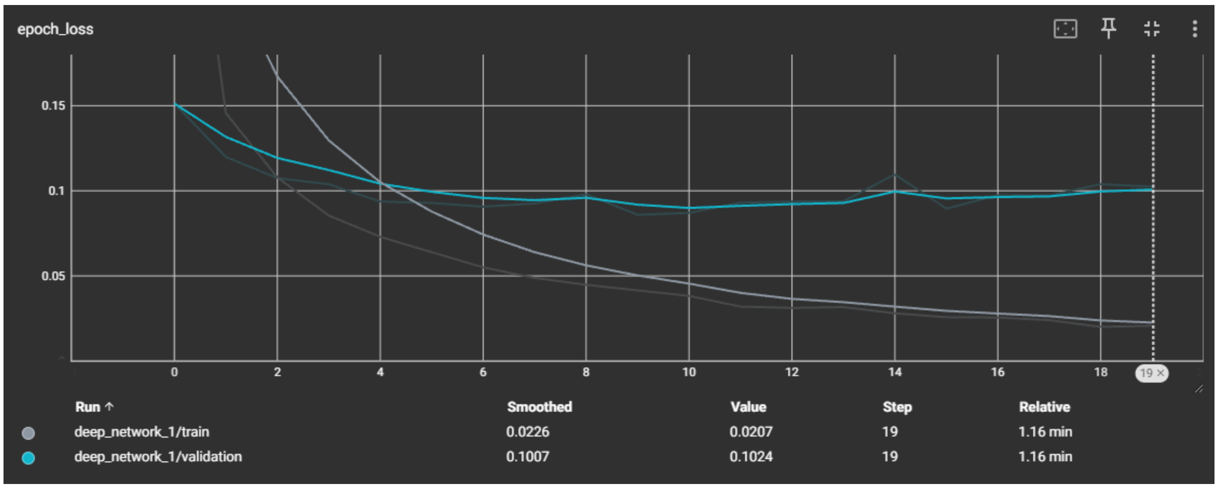
1. First model as it came with notebook

The very first shot was the default architecture I encountered once opened provided notebook – without any modifications.

Provided model consisted of Input layer, 3x Middle layer with 128 neurons and softmax output layer.



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_1/train	0.9922	0.9927	19	1.16 min
●	deep_network_1/validation	0.9759	0.976	19	1.16 min



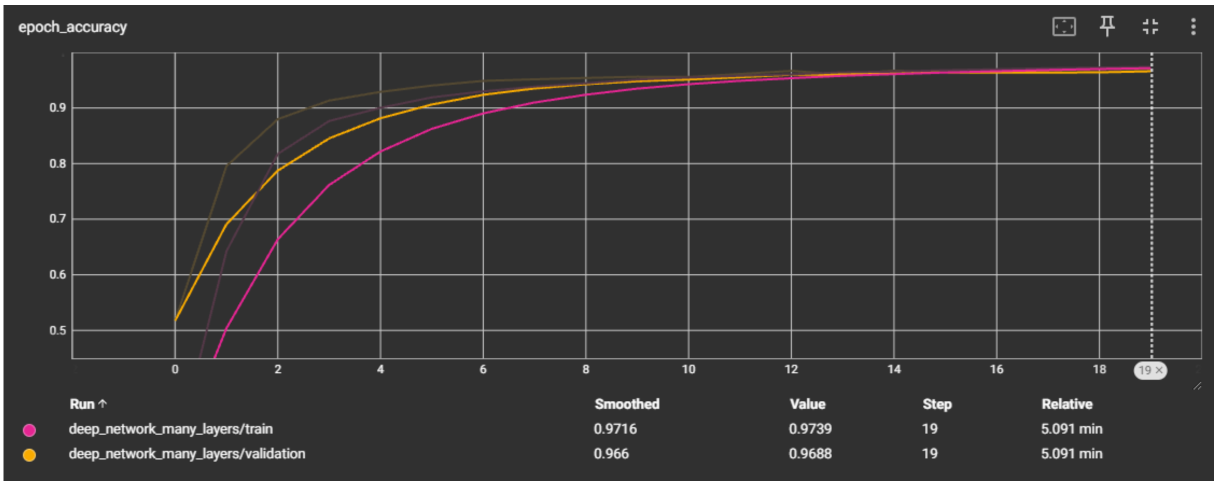
	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_1/train	0.0226	0.0207	19	1.16 min
●	deep_network_1/validation	0.1007	0.1024	19	1.16 min

Validation Accuracy: 0.976

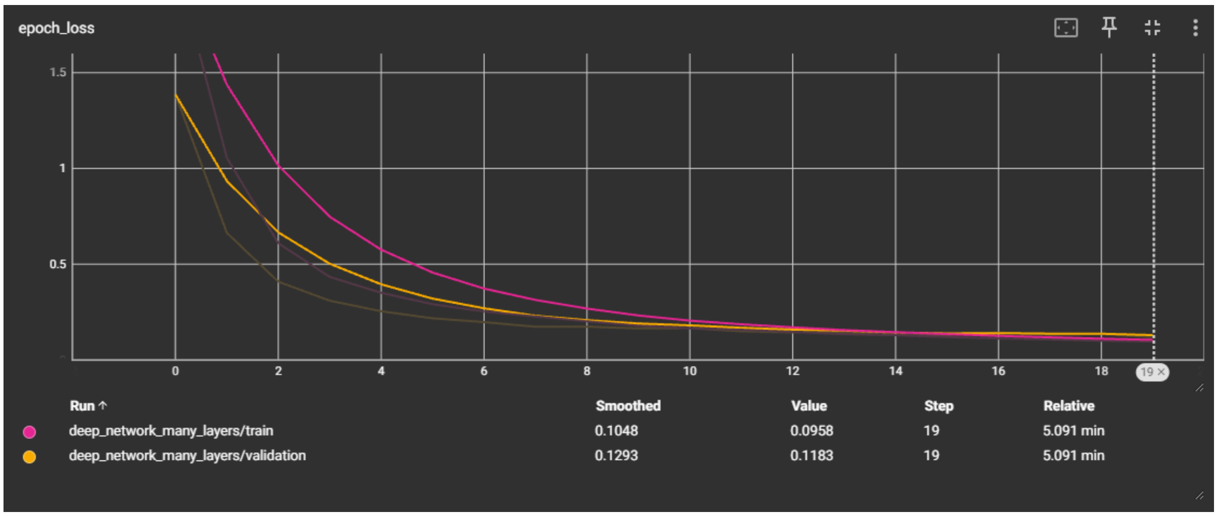
Training Speed: Very fast to train

Comment: This model was reference point for following model architectures.

2. Many layers – 20 extra deeper layers



Run ↑	Smoothed	Value	Step	Relative
● deep_network_many_layers/train	0.9716	0.9739	19	5.091 min
● deep_network_many_layers/validation	0.966	0.9688	19	5.091 min



Run ↑	Smoothed	Value	Step	Relative
● deep_network_many_layers/train	0.1048	0.0958	19	5.091 min
● deep_network_many_layers/validation	0.1293	0.1183	19	5.091 min

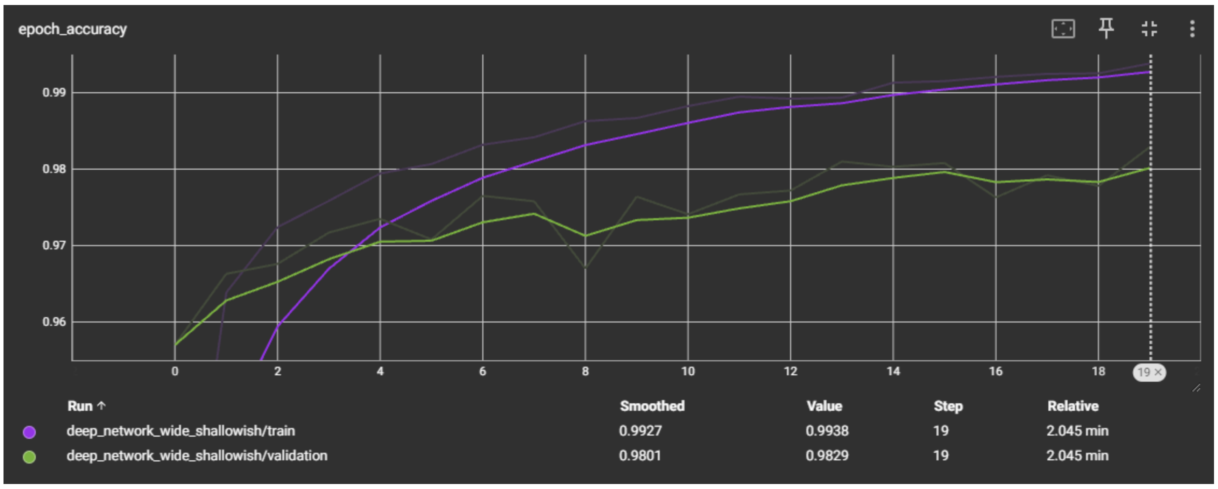
Validation Accuracy: 0.968

Training Speed: 5 minutes

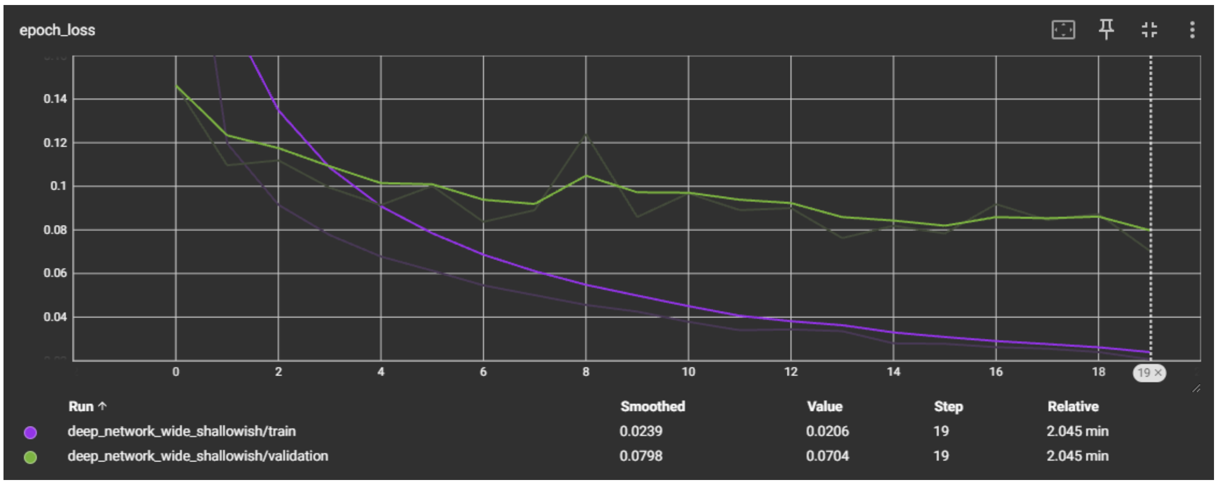
Comment: This model was rather slowly converging with accuracy – not until 10 epochs did we reach acceptable range of values.

3. Wide but Shallow-ish

As for this model I have chosen to use 5 layers but each rather wide – that means the high batch size ranging 256 – 512. For the model I have eventually chosen to consider for this category of architecture I sticked with batch size equal to 512.



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_wide_shallowish/train	0.9927	0.9938	19	2.045 min
●	deep_network_wide_shallowish/validation	0.9801	0.9829	19	2.045 min



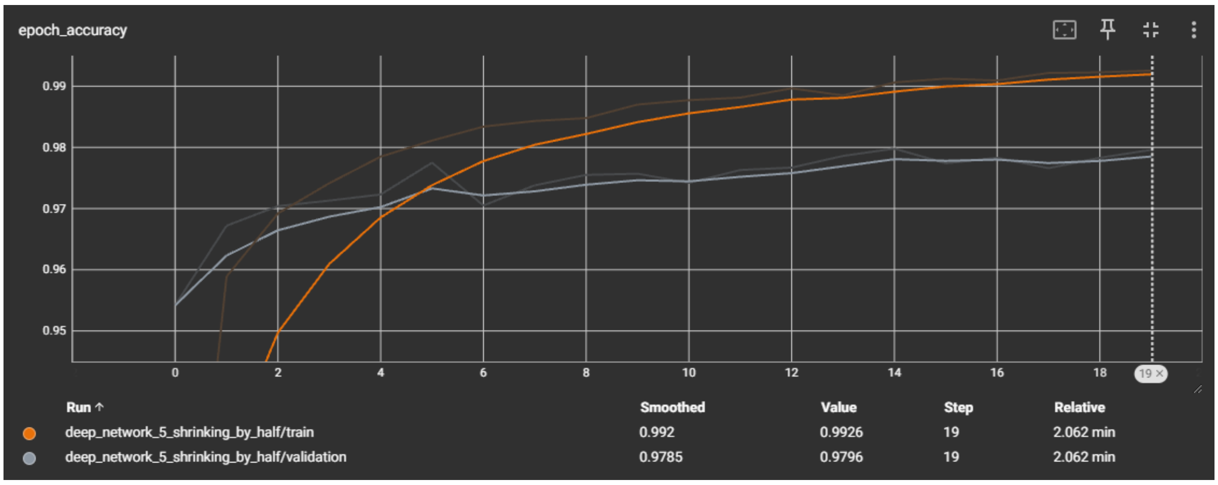
	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_wide_shallowish/train	0.0239	0.0206	19	2.045 min
●	deep_network_wide_shallowish/validation	0.0798	0.0704	19	2.045 min

Validation Accuracy: 0.968

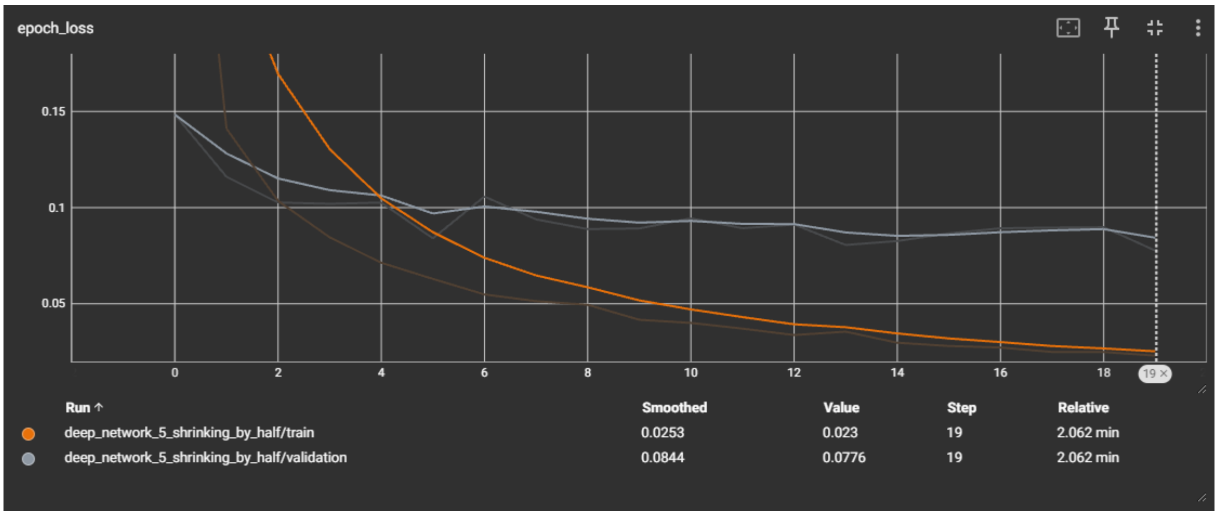
Training Speed: 2 minutes

Comment. One of the best models. Super quick to train.

4. Shrinking by half: 5 layers



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_5_shrinking_by_half/train	0.992	0.9926	19	2.062 min
●	deep_network_5_shrinking_by_half/validation	0.9785	0.9796	19	2.062 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_5_shrinking_by_half/train	0.0253	0.023	19	2.062 min
●	deep_network_5_shrinking_by_half/validation	0.0844	0.0776	19	2.062 min

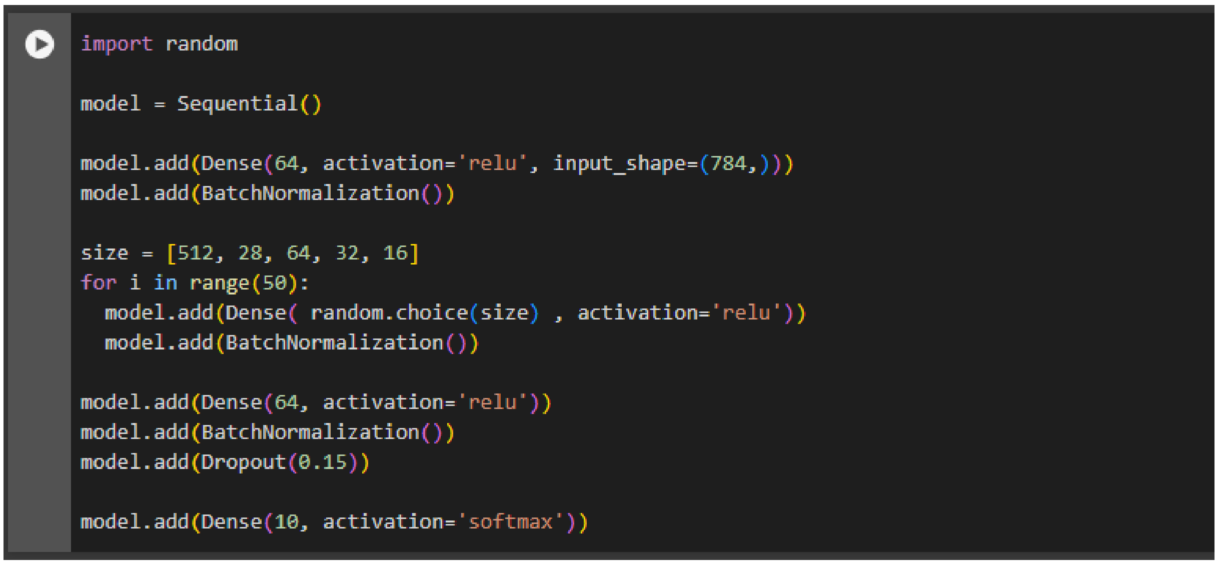
Validation Accuracy: 0.979

Training Speed: 2 minutes

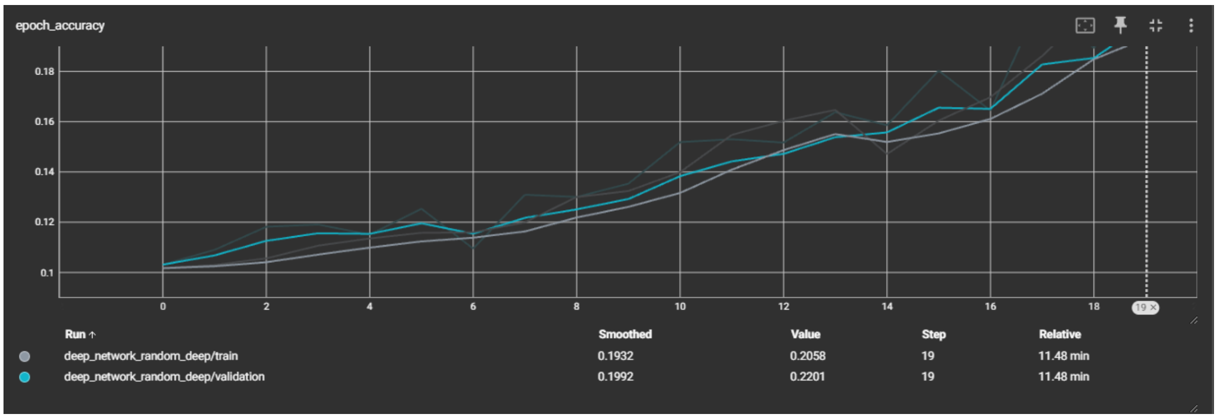
Comment: Pretty good model with high accuracy. Converged very quickly. Epoch loss very small.

5. Random layers’ sizes: 50 Layers

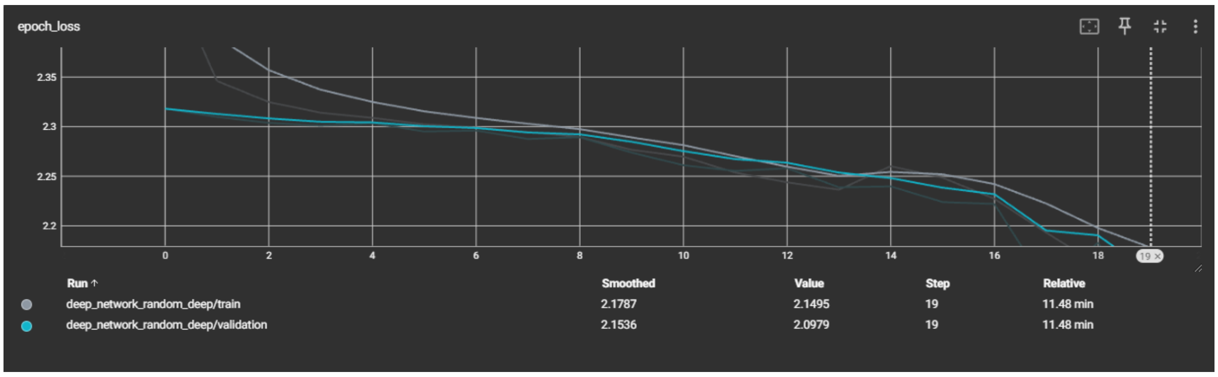
The objective of this model is to evaluate its performance when the sizes of the given layers are determined dynamically based on a normal distribution. We randomly selected batch sizes ranging from 16 to 512 to assess the model's behavior across different batch size configurations.



```
import random

model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

size = [512, 28, 64, 32, 16]
for i in range(50):
  model.add(Dense( random.choice(size) , activation='relu'))
  model.add(BatchNormalization())

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.15))

model.add(Dense(10, activation='softmax'))
```



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_random_deep/train	0.1932	0.2058	19	11.48 min
●	deep_network_random_deep/validation	0.1992	0.2201	19	11.48 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_random_deep/train	2.1787	2.1495	19	11.48 min
●	deep_network_random_deep/validation	2.1536	2.0979	19	11.48 min

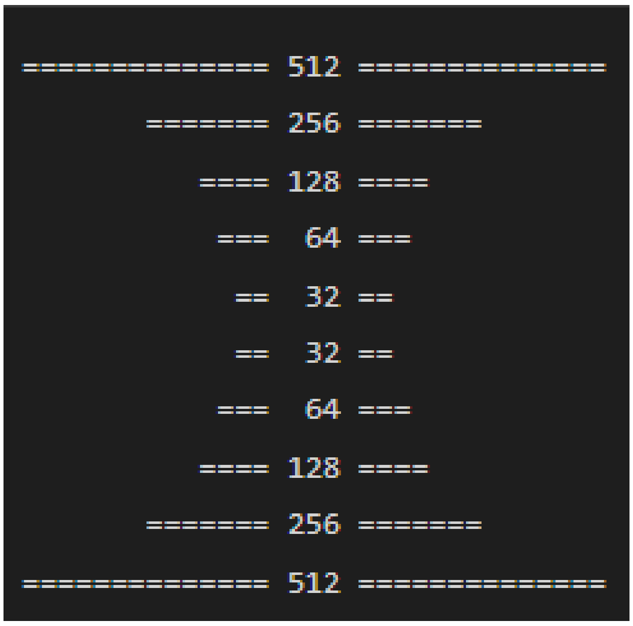
The results were so disappointing that afterwards I gave up any hopes for random element in depth and size of layers selection. Not only was it unbearably slow to train but also it was underperforming very shallow models.
Taking into consideration the cost of training models It is definitely not worth it given the fact I tried to make it not as big and maneuvering with the layers sizes as well.

Validation Accuracy: 0.20

Training Speed: 11 minutes

Comment:
Not to be considered great deal – Avoid random layers' sizes. Model's accuracy is so bad I strongly discourage anyone perusing randomness that matter.

6. Big – Small – Big

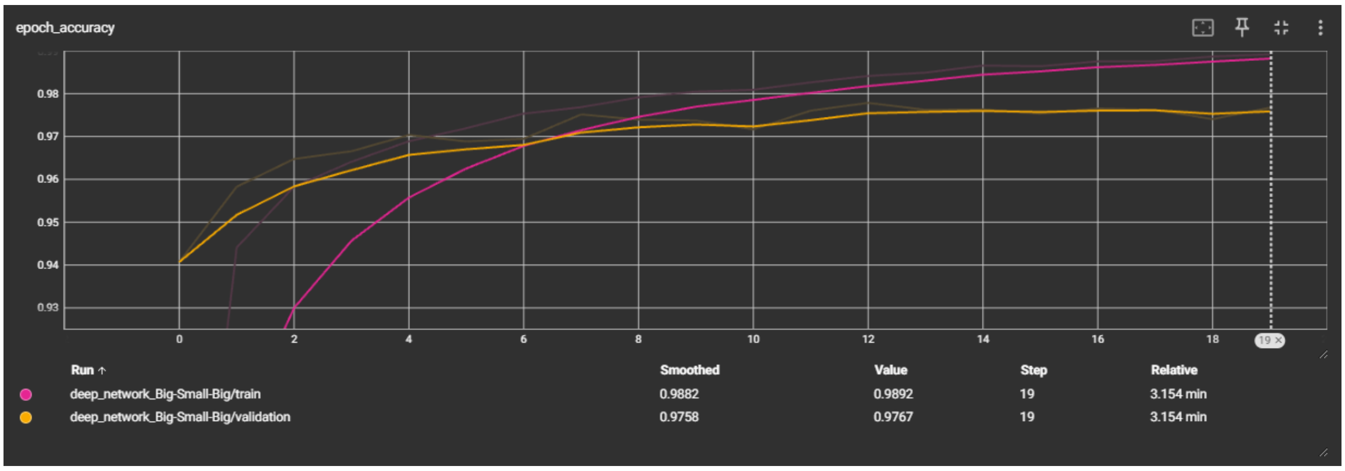


```
============== 512 ==============
       ======= 256 =======
          ==== 128 ====
           ===  64 ===
            ==  32 ==
            ==  32 ==
           ===  64 ===
          ==== 128 ====
       ======= 256 =======
============== 512 ==============
```

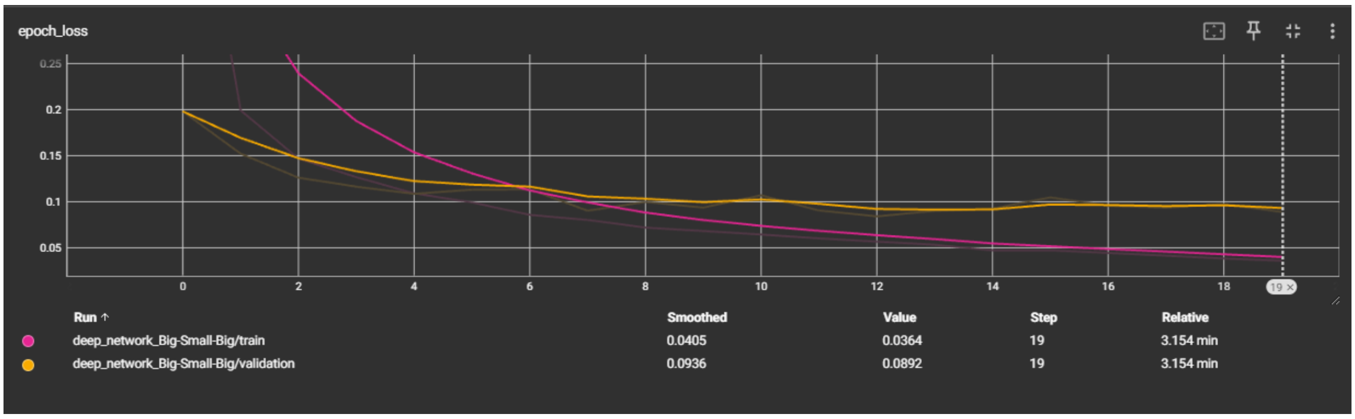
Validation Accuracy: 0.9767

Training Speed: 3 minutes

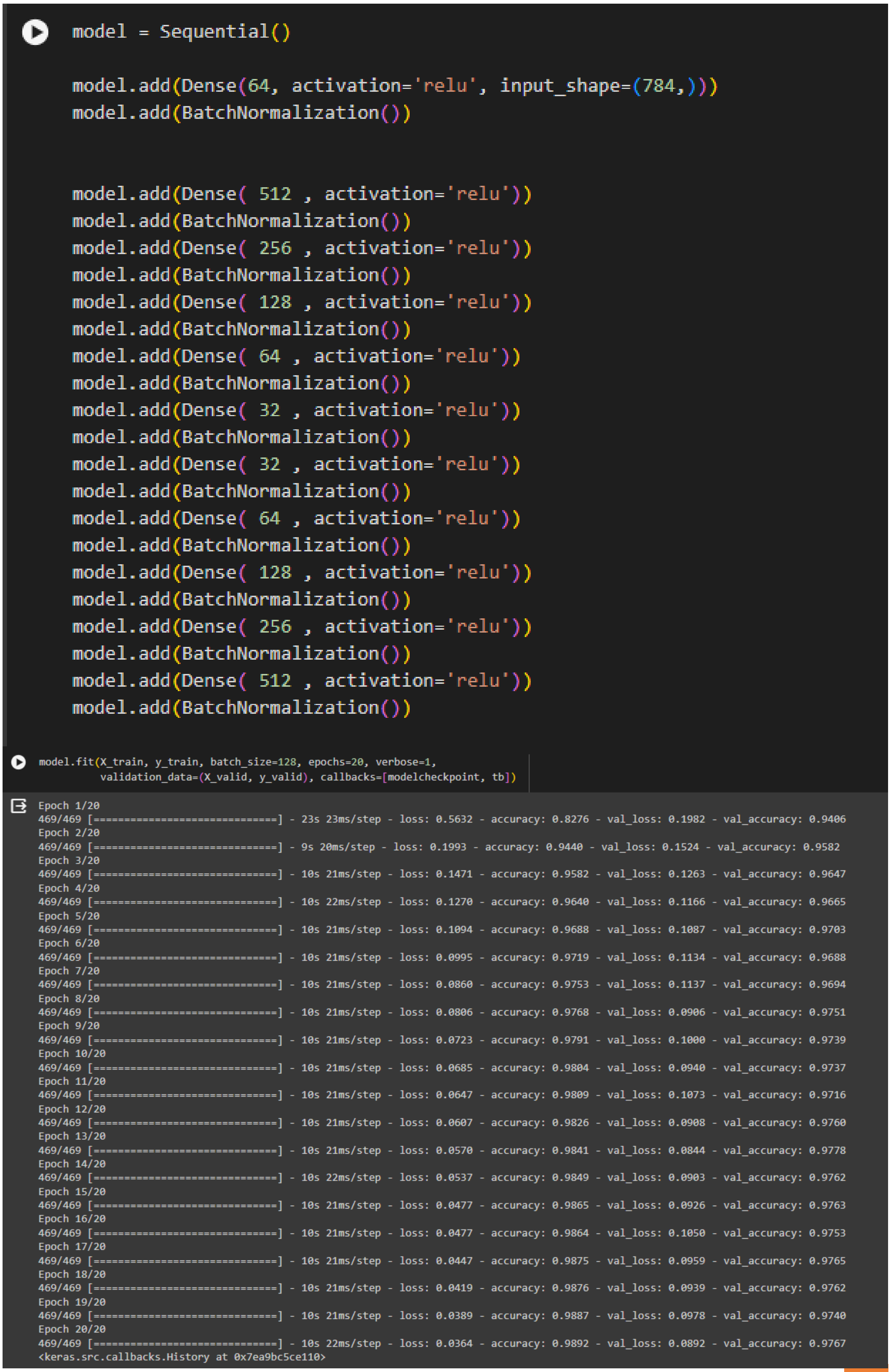
Comment: Pretty good model with high accuracy. Converged very quickly. Epoch loss very small. Quick to train. Additionally simple architecture.



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_Big-Small-Big/train	0.9882	0.9892	19	3.154 min
●	deep_network_Big-Small-Big/validation	0.9758	0.9767	19	3.154 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_Big-Small-Big/train	0.0405	0.0364	19	3.154 min
●	deep_network_Big-Small-Big/validation	0.0936	0.0892	19	3.154 min

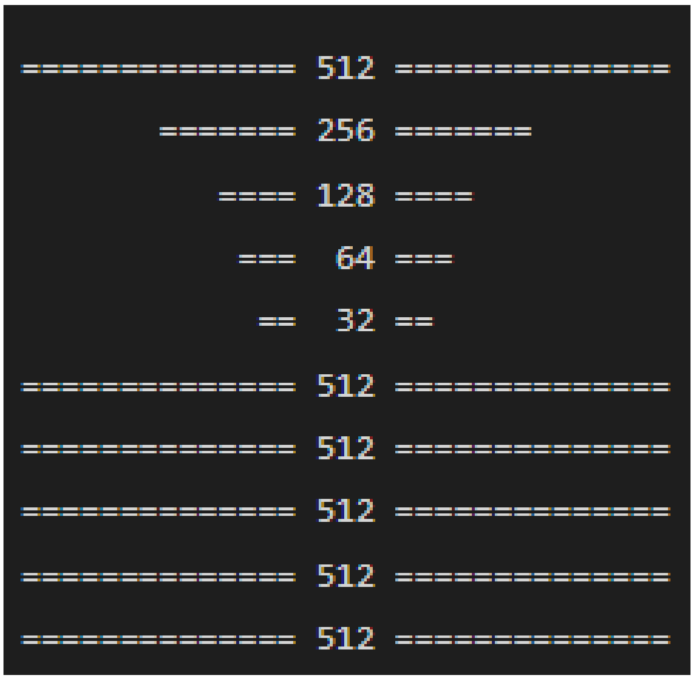


```python
model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())


model.add(Dense( 512 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 256 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 128 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 64 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 32 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 32 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 64 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 128 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 256 , activation='relu'))
model.add(BatchNormalization())
model.add(Dense( 512 , activation='relu'))
model.add(BatchNormalization())
```

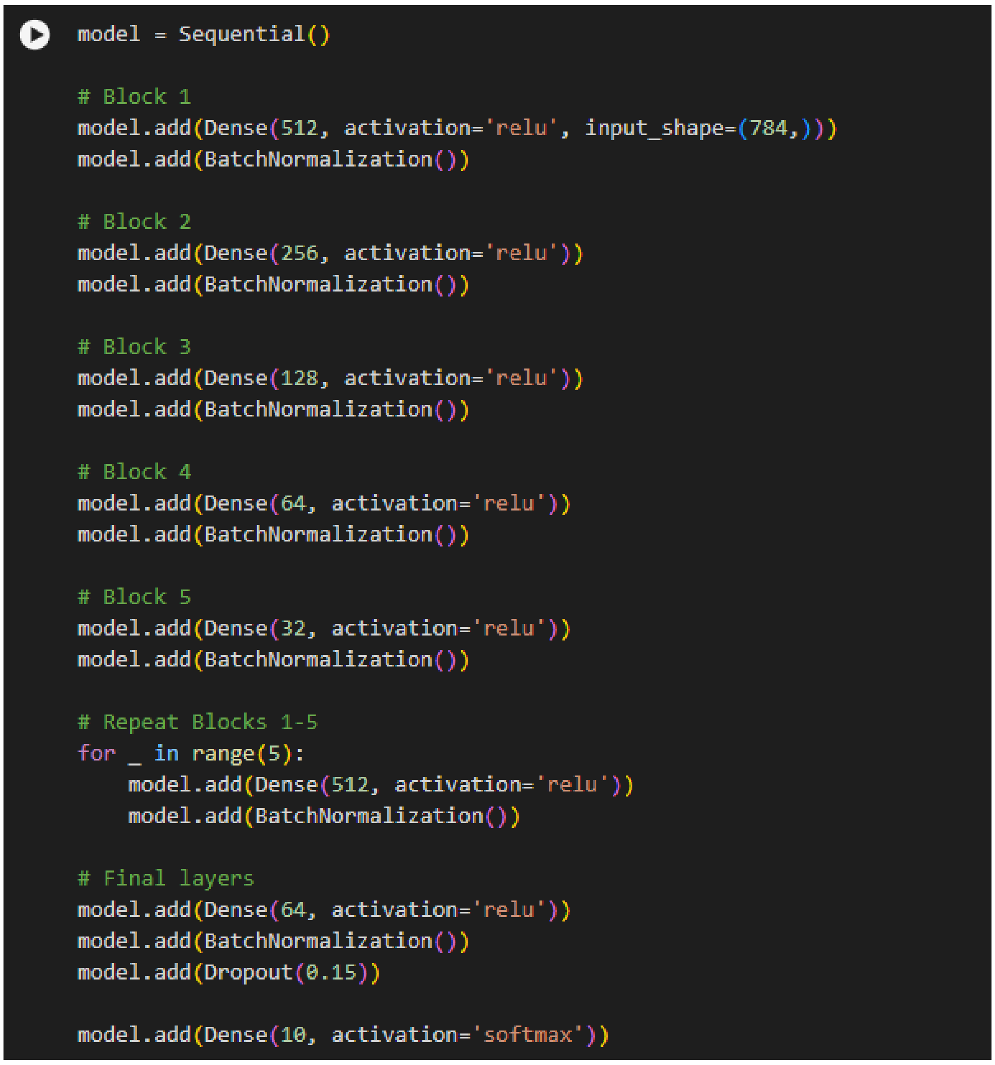
```python
model.fit(X_train, y_train, batch_size=128, epochs=20, verbose=1,
          validation_data=(X_valid, y_valid), callbacks=[modelcheckpoint, tb])
```

```
Epoch 1/20
469/469 [==============================] - 23s 23ms/step - loss: 0.5632 - accuracy: 0.8276 - val_loss: 0.1982 - val_accuracy: 0.9406
Epoch 2/20
469/469 [==============================] - 9s 20ms/step - loss: 0.1993 - accuracy: 0.9440 - val_loss: 0.1524 - val_accuracy: 0.9582
Epoch 3/20
469/469 [==============================] - 10s 21ms/step - loss: 0.1471 - accuracy: 0.9582 - val_loss: 0.1263 - val_accuracy: 0.9647
Epoch 4/20
469/469 [==============================] - 10s 22ms/step - loss: 0.1270 - accuracy: 0.9640 - val_loss: 0.1166 - val_accuracy: 0.9665
Epoch 5/20
469/469 [==============================] - 10s 21ms/step - loss: 0.1094 - accuracy: 0.9688 - val_loss: 0.1087 - val_accuracy: 0.9703
Epoch 6/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0995 - accuracy: 0.9719 - val_loss: 0.1134 - val_accuracy: 0.9688
Epoch 7/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0860 - accuracy: 0.9753 - val_loss: 0.1137 - val_accuracy: 0.9694
Epoch 8/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0806 - accuracy: 0.9768 - val_loss: 0.0906 - val_accuracy: 0.9751
Epoch 9/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0723 - accuracy: 0.9791 - val_loss: 0.1000 - val_accuracy: 0.9739
Epoch 10/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0685 - accuracy: 0.9804 - val_loss: 0.0940 - val_accuracy: 0.9737
Epoch 11/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0647 - accuracy: 0.9809 - val_loss: 0.1073 - val_accuracy: 0.9716
Epoch 12/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0607 - accuracy: 0.9826 - val_loss: 0.0908 - val_accuracy: 0.9760
Epoch 13/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0570 - accuracy: 0.9841 - val_loss: 0.0844 - val_accuracy: 0.9778
Epoch 14/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0537 - accuracy: 0.9849 - val_loss: 0.0903 - val_accuracy: 0.9762
Epoch 15/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0477 - accuracy: 0.9865 - val_loss: 0.0926 - val_accuracy: 0.9763
Epoch 16/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0477 - accuracy: 0.9864 - val_loss: 0.1050 - val_accuracy: 0.9753
Epoch 17/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0447 - accuracy: 0.9875 - val_loss: 0.0959 - val_accuracy: 0.9765
Epoch 18/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0419 - accuracy: 0.9876 - val_loss: 0.0939 - val_accuracy: 0.9762
Epoch 19/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0389 - accuracy: 0.9887 - val_loss: 0.0978 - val_accuracy: 0.9740
Epoch 20/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0364 - accuracy: 0.9892 - val_loss: 0.0892 - val_accuracy: 0.9767
<keras.src.callbacks.History at 0x7ea9bc5ce110>
```

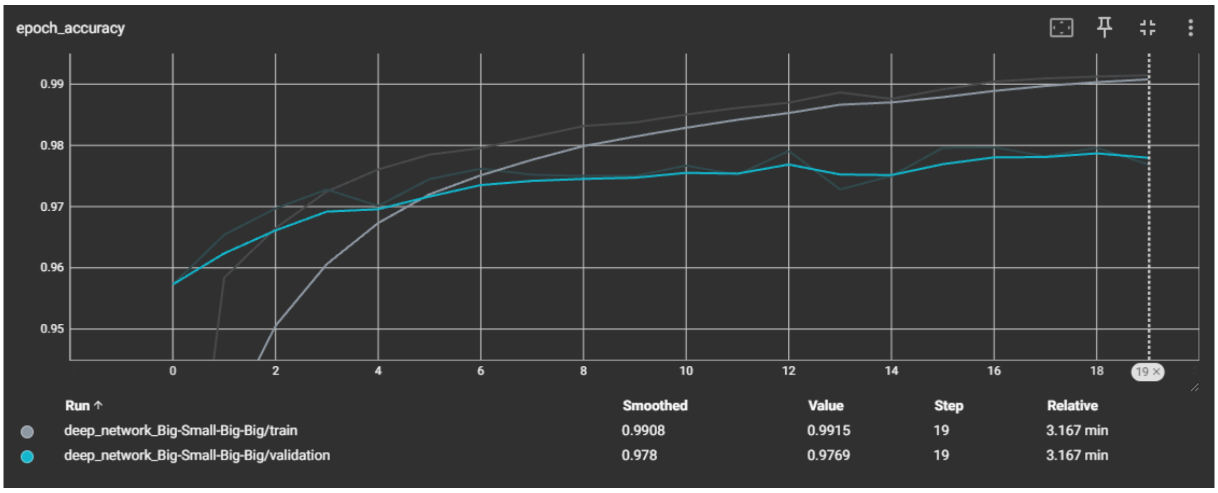
7. Big – Small – Big … Big



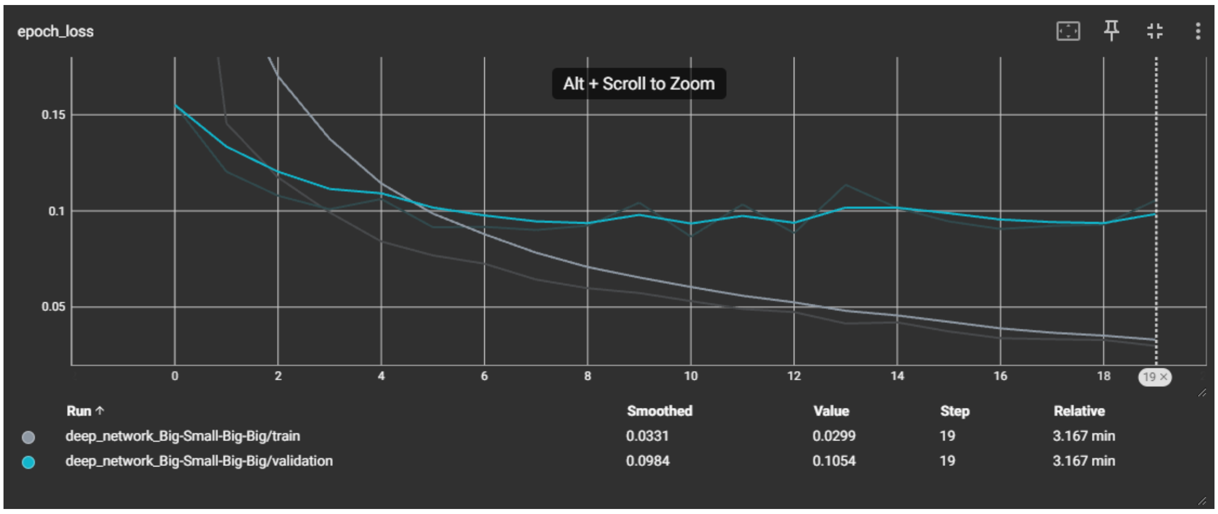
```
============== 512 ==============
       ======= 256 =======
          ==== 128 ====
           ===  64 ===
            ==  32 ==
============== 512 ==============
============== 512 ==============
============== 512 ==============
============== 512 ==============
============== 512 ==============
```



```python
model = Sequential()

# Block 1
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

# Block 2
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())

# Block 3
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())

# Block 4
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())

# Block 5
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())

# Repeat Blocks 1-5
for _ in range(5):
    model.add(Dense(512, activation='relu'))
    model.add(BatchNormalization())

# Final layers
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.15))

model.add(Dense(10, activation='softmax'))
```



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_Big-Small-Big-Big/train	0.9908	0.9915	19	3.167 min
●	deep_network_Big-Small-Big-Big/validation	0.978	0.9769	19	3.167 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_Big-Small-Big-Big/train	0.0331	0.0299	19	3.167 min
●	deep_network_Big-Small-Big-Big/validation	0.0984	0.1054	19	3.167 min

Validation Accuracy: 0.9769

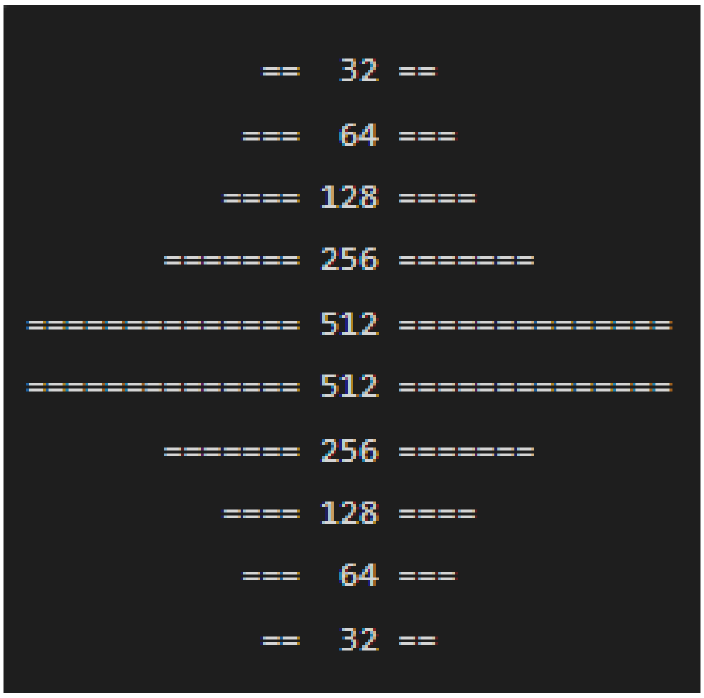
Training Speed: 3.16 minutes

Comment:

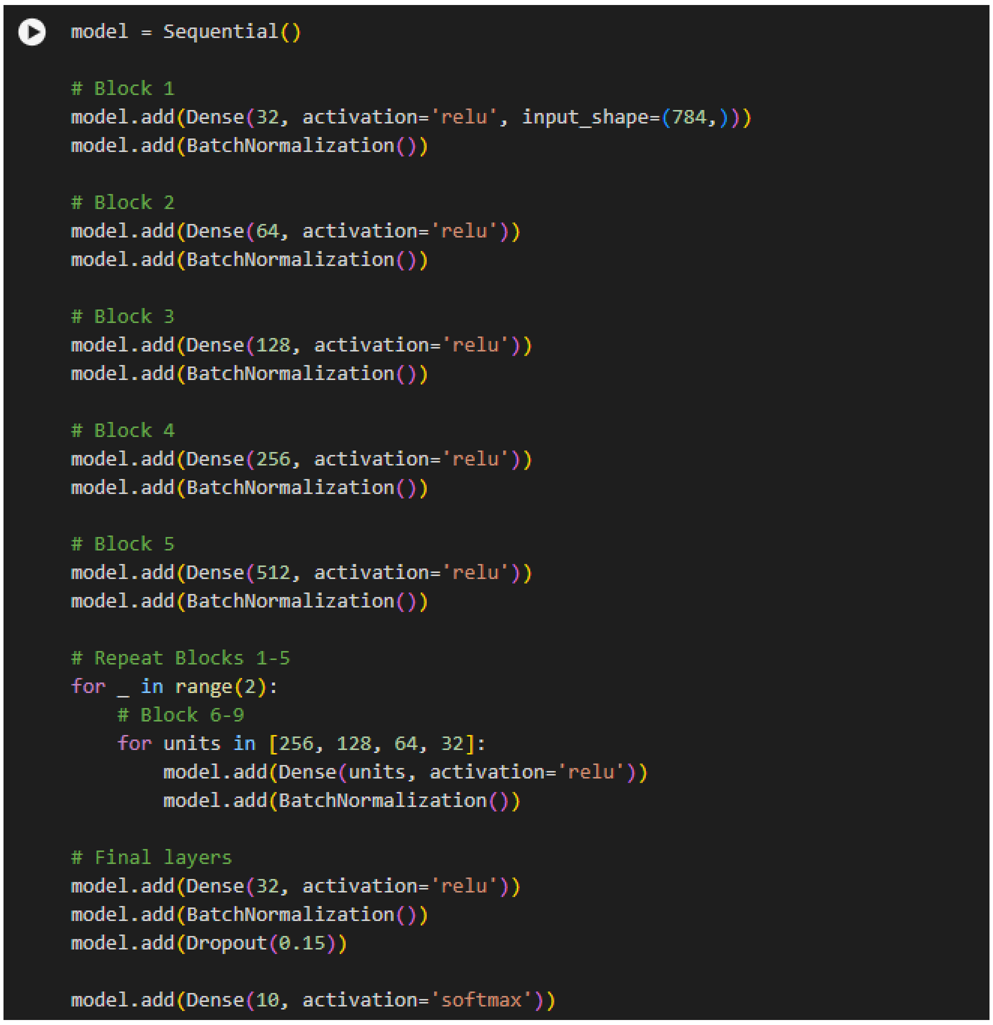
Taken into consideration previous successes with narrowing layer sizes architectures I came to conclusion that this must be the way to go and came up with an idea to mix together both architectures “Big – Small – Big” with “Wide Shallow-ish”.

The model is not to deep and can be trained quickly. Relatively high accuracy.

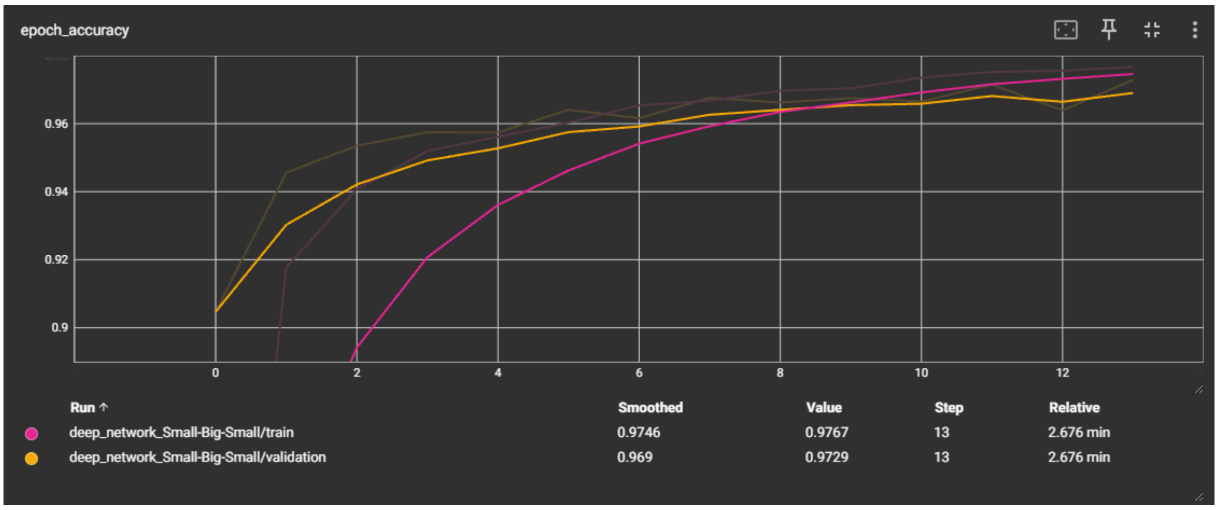
8. Small – Big – Small



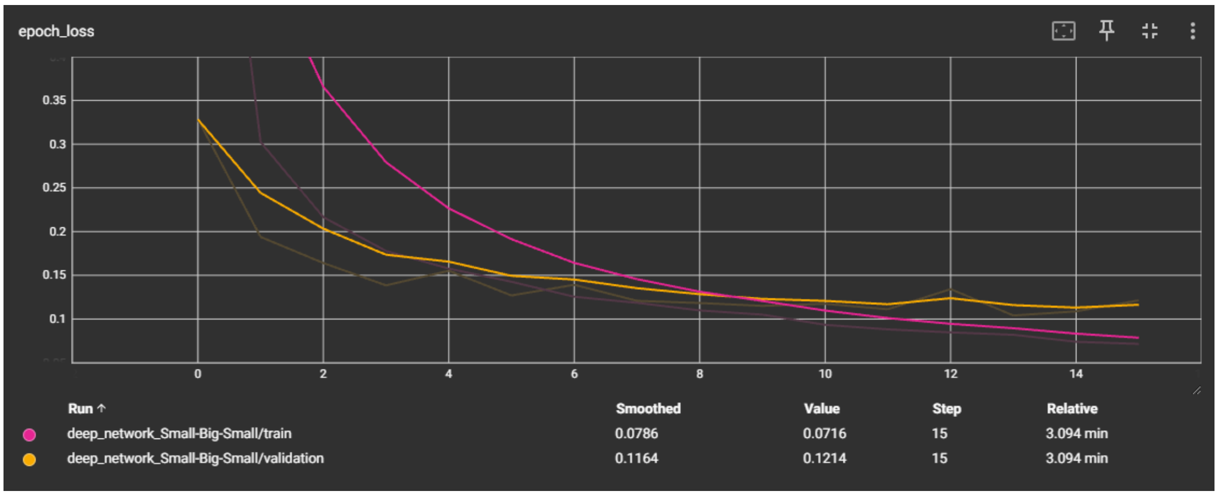
```
          ==  32 ==
         ===  64 ===
        ==== 128 ====
     ======= 256 =======
============== 512 ==============
============== 512 ==============
     ======= 256 =======
        ==== 128 ====
         ===  64 ===
          ==  32 ==
```



```python
model = Sequential()

# Block 1
model.add(Dense(32, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

# Block 2
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())

# Block 3
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())

# Block 4
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())

# Block 5
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())

# Repeat Blocks 1-5
for _ in range(2):
    # Block 6-9
    for units in [256, 128, 64, 32]:
        model.add(Dense(units, activation='relu'))
        model.add(BatchNormalization())

# Final layers
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.15))

model.add(Dense(10, activation='softmax'))
```



Run ↑	Smoothed	Value	Step	Relative
● deep_network_Small-Big-Small/train	0.9746	0.9767	13	2.676 min
● deep_network_Small-Big-Small/validation	0.969	0.9729	13	2.676 min



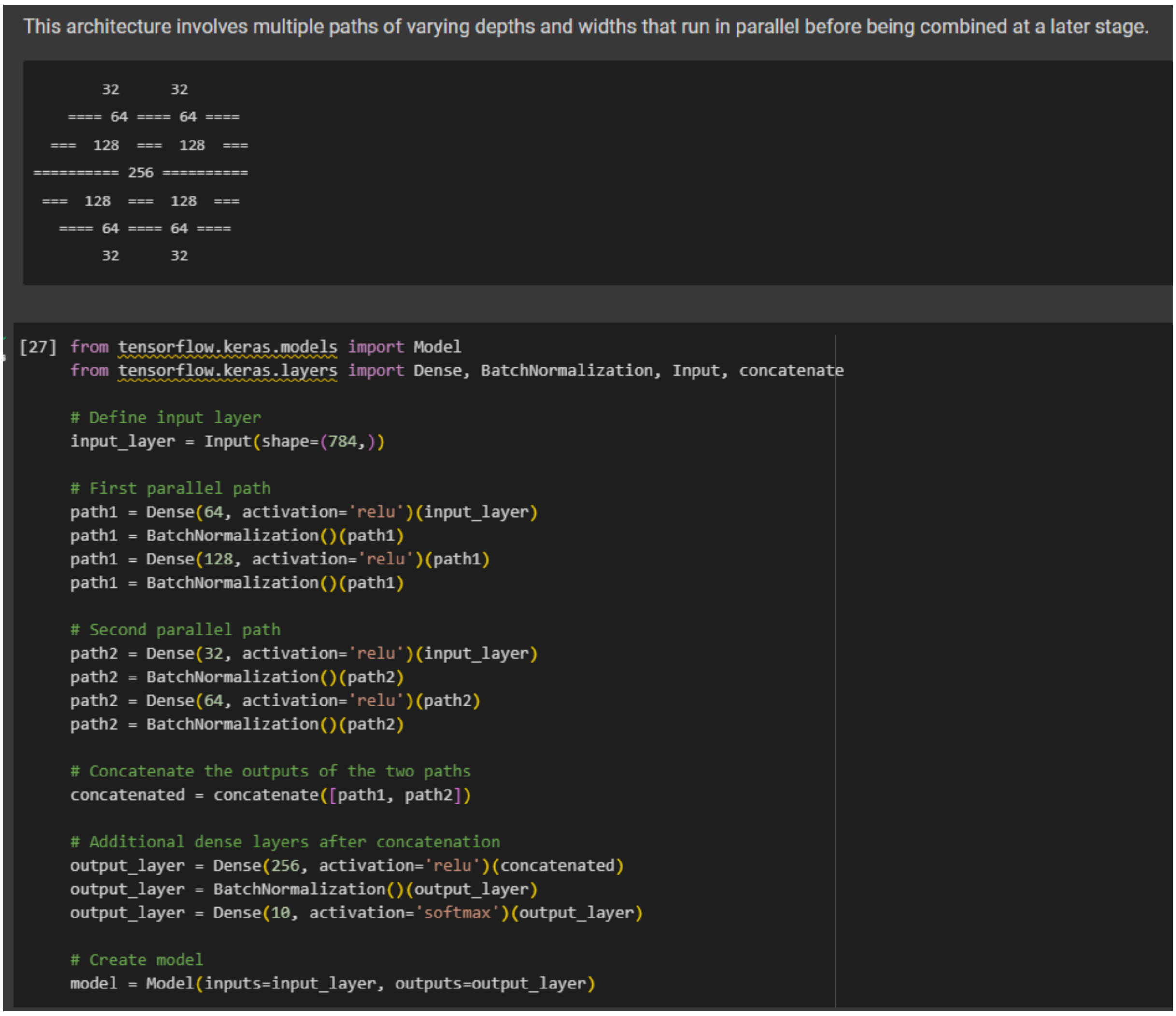
Run ↑	Smoothed	Value	Step	Relative
● deep_network_Small-Big-Small/train	0.0786	0.0716	15	3.094 min
● deep_network_Small-Big-Small/validation	0.1164	0.1214	15	3.094 min

Validation Accuracy: 0.9729

Training Speed: Relatively quick to train

Comment: Pretty good model with high accuracy. Converged very quickly. Epoch loss very small. Quick to train. There are some differences between training and testing data.

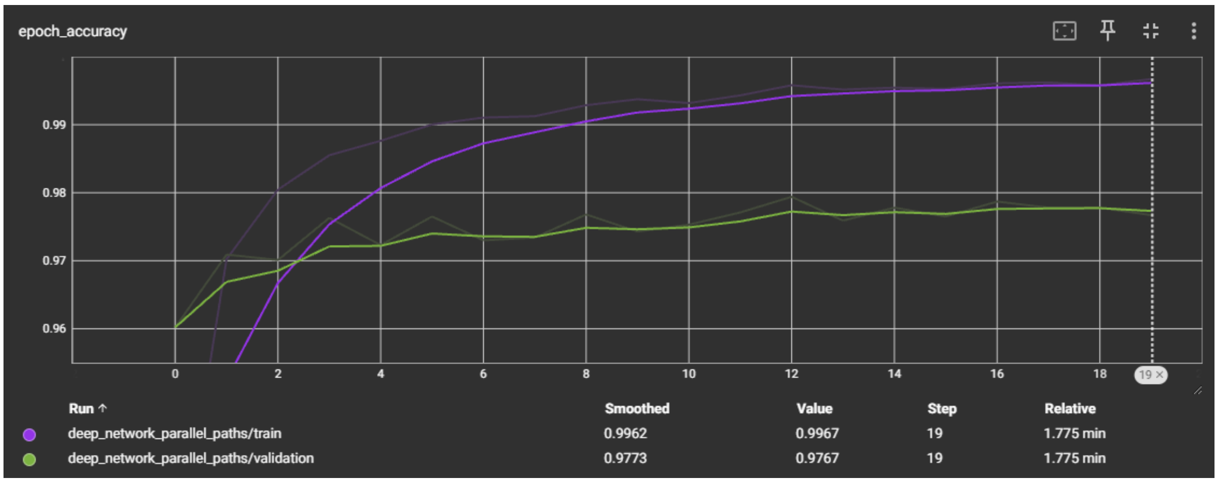
9. Parallel Paths



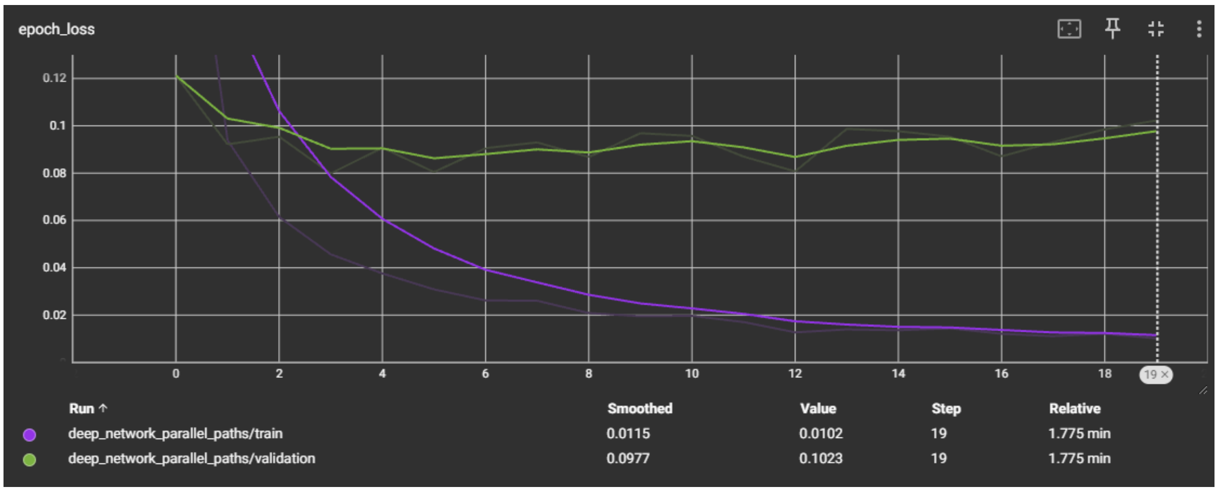
This architecture involves multiple paths of varying depths and widths that run in parallel before being combined at a later stage.

```
        32       32
    ==== 64 ==== 64 ====
  ===  128  ===  128  ===
========== 256 ==========
 ===  128  ===  128  ===
   ==== 64 ==== 64 ====
        32       32
```

```
[27] from tensorflow.keras.models import Model
     from tensorflow.keras.layers import Dense, BatchNormalization, Input, concatenate

     # Define input layer
     input_layer = Input(shape=(784,))

     # First parallel path
     path1 = Dense(64, activation='relu')(input_layer)
     path1 = BatchNormalization()(path1)
     path1 = Dense(128, activation='relu')(path1)
     path1 = BatchNormalization()(path1)

     # Second parallel path
     path2 = Dense(32, activation='relu')(input_layer)
     path2 = BatchNormalization()(path2)
     path2 = Dense(64, activation='relu')(path2)
     path2 = BatchNormalization()(path2)

     # Concatenate the outputs of the two paths
     concatenated = concatenate([path1, path2])

     # Additional dense layers after concatenation
     output_layer = Dense(256, activation='relu')(concatenated)
     output_layer = BatchNormalization()(output_layer)
     output_layer = Dense(10, activation='softmax')(output_layer)

     # Create model
     model = Model(inputs=input_layer, outputs=output_layer)
```



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_parallel_paths/train	0.9962	0.9967	19	1.775 min
●	deep_network_parallel_paths/validation	0.9773	0.9767	19	1.775 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_parallel_paths/train	0.0115	0.0102	19	1.775 min
●	deep_network_parallel_paths/validation	0.0977	0.1023	19	1.775 min

Validation Accuracy: 0.9767

Training Speed: Relatively quick to train

Comment:

In this model I tried something new – Introducing parallel paths.

In a parallel model architecture, the input signal is typically split into multiple branches, with each branch processing a copy of the input data independently. This splitting of the input signal allows different branches of the model to focus on learning different aspects or representations of the input data simultaneously.

Each branch operates on its own copy of the input signal, applying its own set of layers and transformations. These branches can learn to extract different features or representations from the input data in parallel. Once the processing within each branch is complete, the outputs are merged together using merge layers or other mechanisms to combine the information learned from different branches.

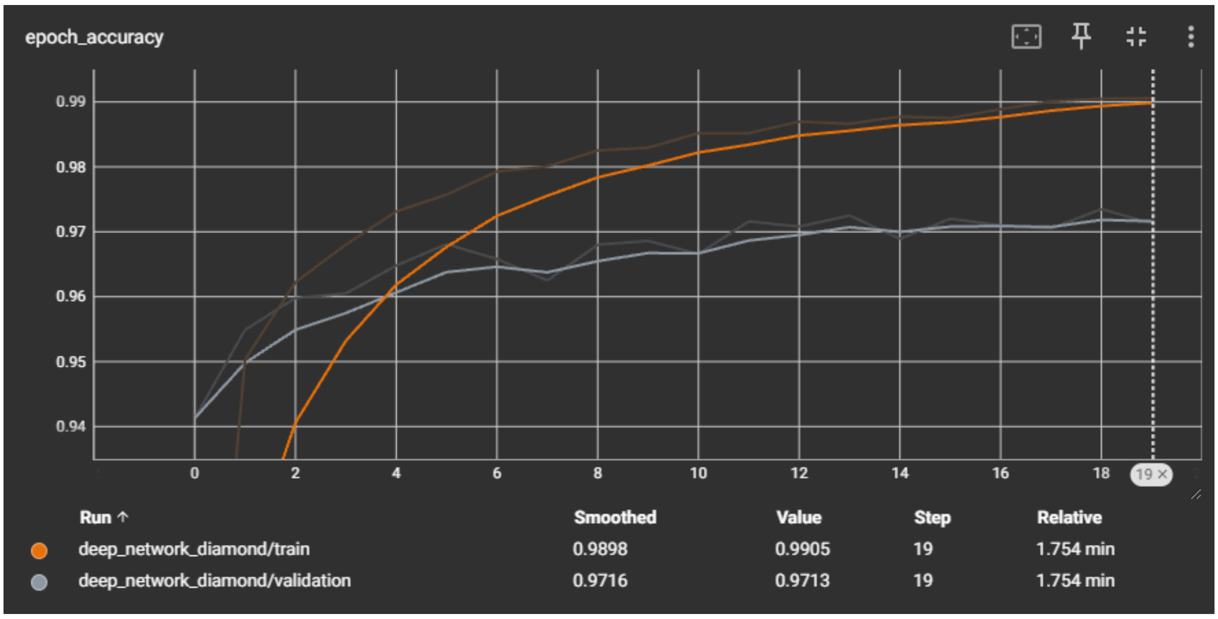
By splitting the input signal into multiple branches, parallel models can effectively capture diverse information and extract complementary features from the input data. This parallel processing can lead to improved model performance and efficiency, especially in tasks where the input data contains complex and diverse patterns or structures.

In this parallel model:

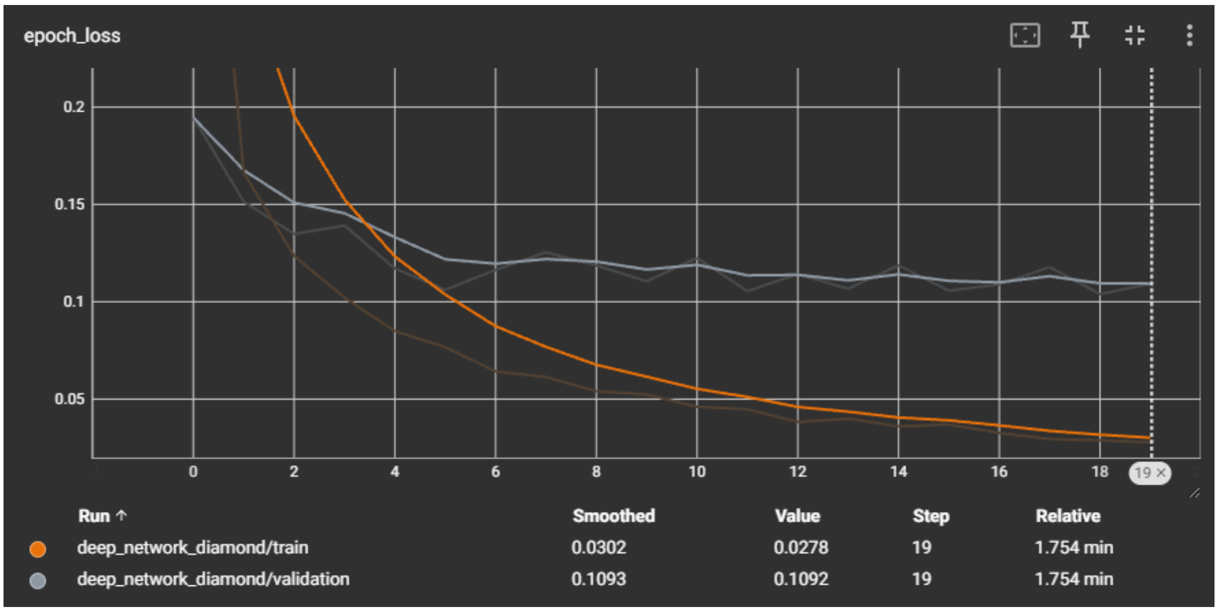
1. **Input Layer**: The input signal, representing your data, is fed into the input layer.
2. **Parallel Branches**:
   - There are four sets of parallel branches, each with two branches.
   - Each set consists of two branches processing the input independently.
3. **Processing Within Branches**:
   - Each branch processes the input signal independently through its own set of layers.
   - The layers within each branch have the specified number of neurons.
   - For example, in the first set of branches (32 and 32), the input signal is processed through two branches with 32 neurons each.
4. **Merge and Further Processing**:
   - After processing through the layers in each branch, the outputs from corresponding branches are merged together.
   - This merging process combines the information learned from both branches within each set.
5. **Output Layer**:
   - Finally, the merged outputs from all sets are further processed through subsequent layers.
   - The model might have additional dense layers or operations after the merging of parallel branches.
   - Eventually, the signal reaches the output layer for the final prediction or classification.

So, the signal flows independently through multiple parallel branches, with each branch learning different representations of the input data. These representations are then combined to form a comprehensive understanding of the input, aiding in the final prediction or classification task.

10. Diamond



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_diamond/train	0.9898	0.9905	19	1.754 min
●	deep_network_diamond/validation	0.9716	0.9713	19	1.754 min

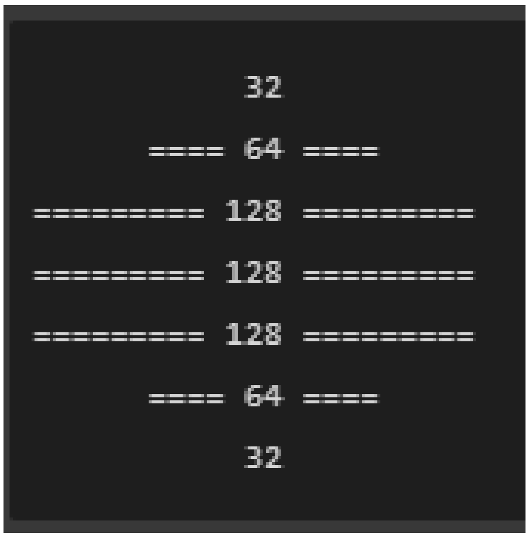


	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_diamond/train	0.0302	0.0278	19	1.754 min
●	deep_network_diamond/validation	0.1093	0.1092	19	1.754 min

Validation Accuracy: 0.9713

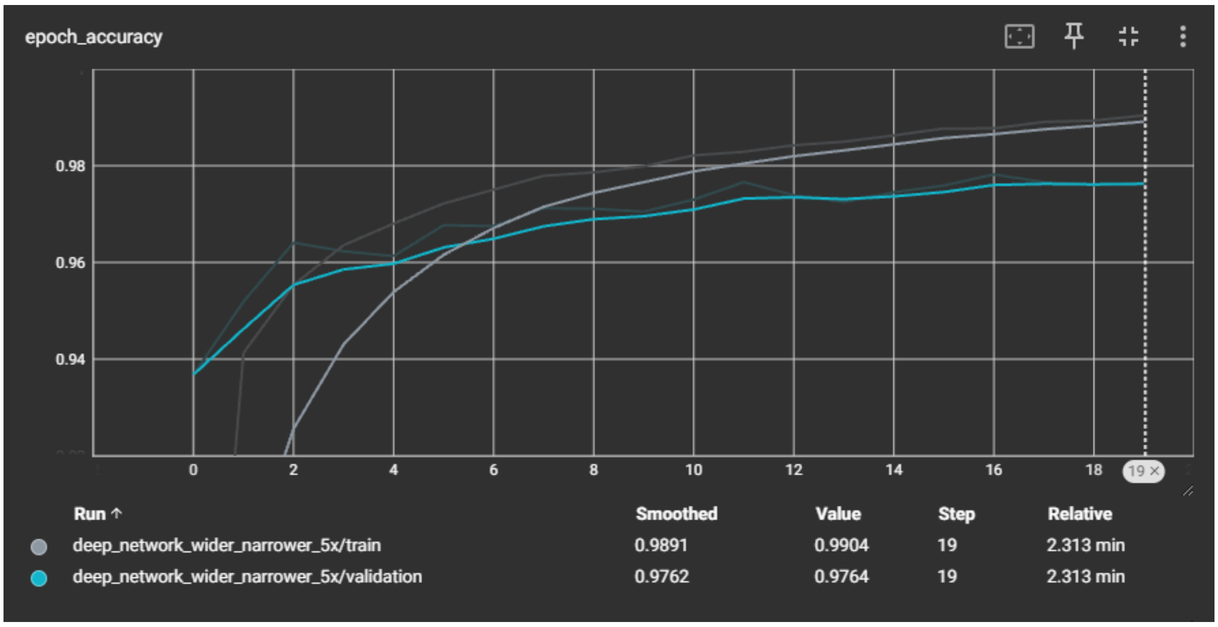
Training Speed: Really, Really Quick (about 1-2 minutes)

Comment: Pretty good model with mid/high accuracy. Converged very quickly. Epoch loss very small. Quick to train. There are some significant differences between training and testing accuracies.

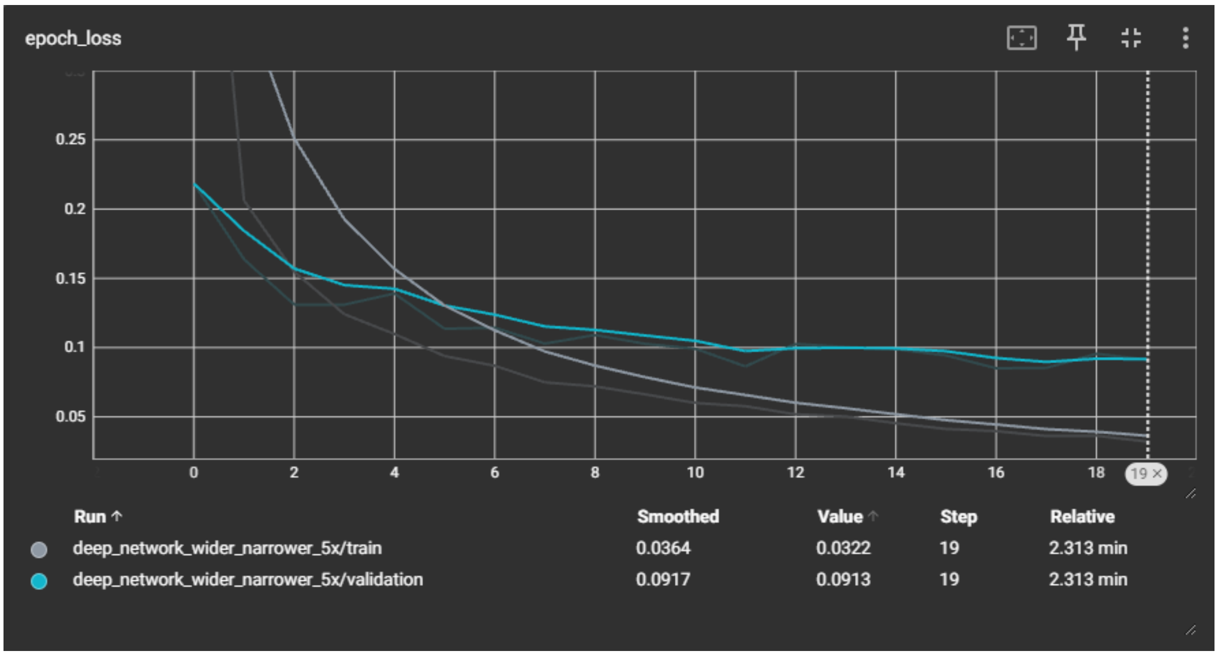


```
           32
      ==== 64 ====
========= 128 =========
========= 128 =========
========= 128 =========
      ==== 64 ====
           32
```

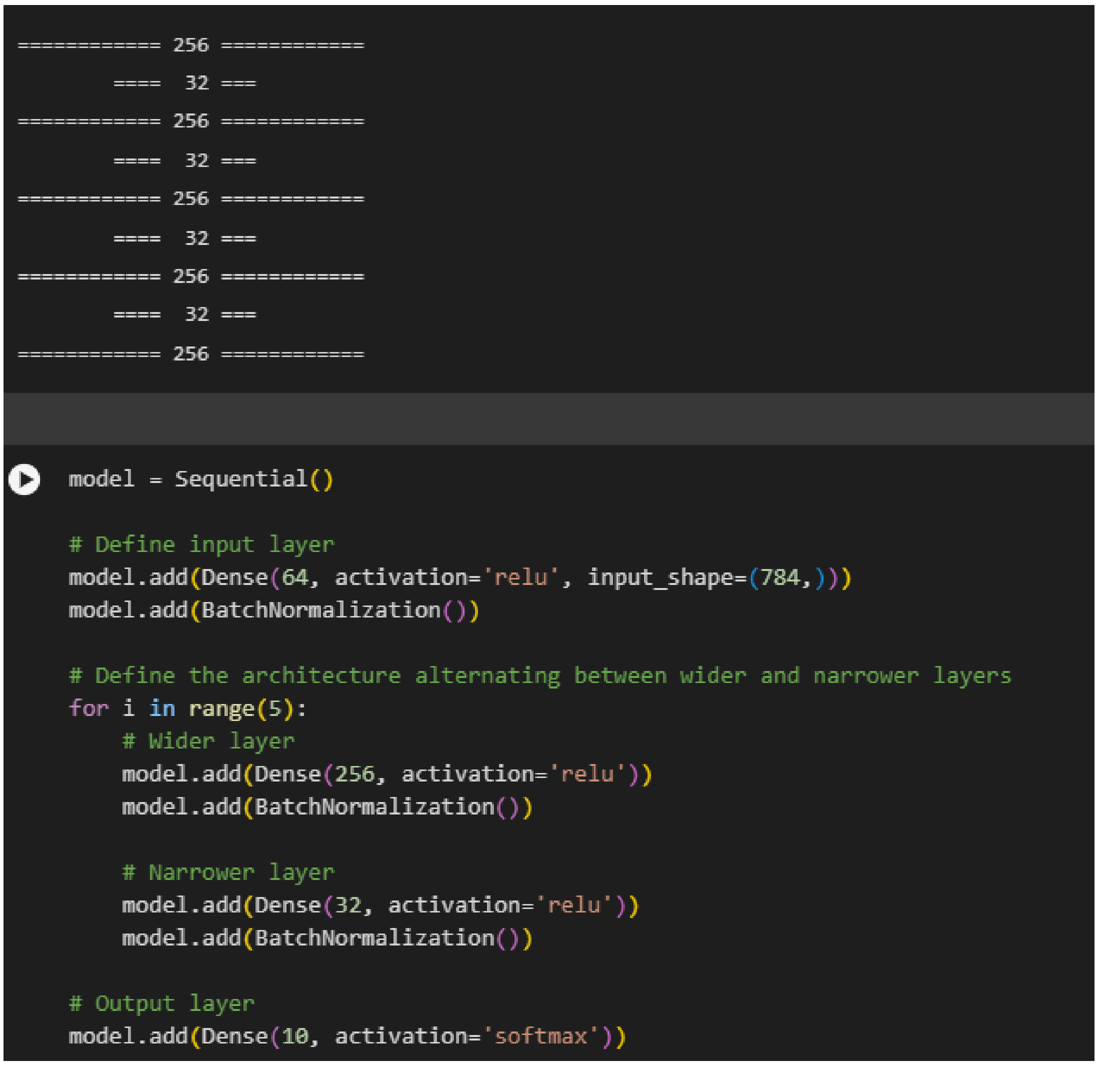
11. Wide – super narrow 5x



Run ↑	Smoothed	Value	Step	Relative
● deep_network_wider_narrower_5x/train	0.9891	0.9904	19	2.313 min
● deep_network_wider_narrower_5x/validation	0.9762	0.9764	19	2.313 min



Run ↑	Smoothed	Value ↑	Step	Relative
● deep_network_wider_narrower_5x/train	0.0364	0.0322	19	2.313 min
● deep_network_wider_narrower_5x/validation	0.0917	0.0913	19	2.313 min



```
============ 256 ============
        ====  32 ===
============ 256 ============
        ====  32 ===
============ 256 ============
        ====  32 ===
============ 256 ============
        ====  32 ===
============ 256 ============
```

```python
model = Sequential()

# Define input layer
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

# Define the architecture alternating between wider and narrower layers
for i in range(5):
    # Wider layer
    model.add(Dense(256, activation='relu'))
    model.add(BatchNormalization())

    # Narrower layer
    model.add(Dense(32, activation='relu'))
    model.add(BatchNormalization())

# Output layer
model.add(Dense(10, activation='softmax'))
```

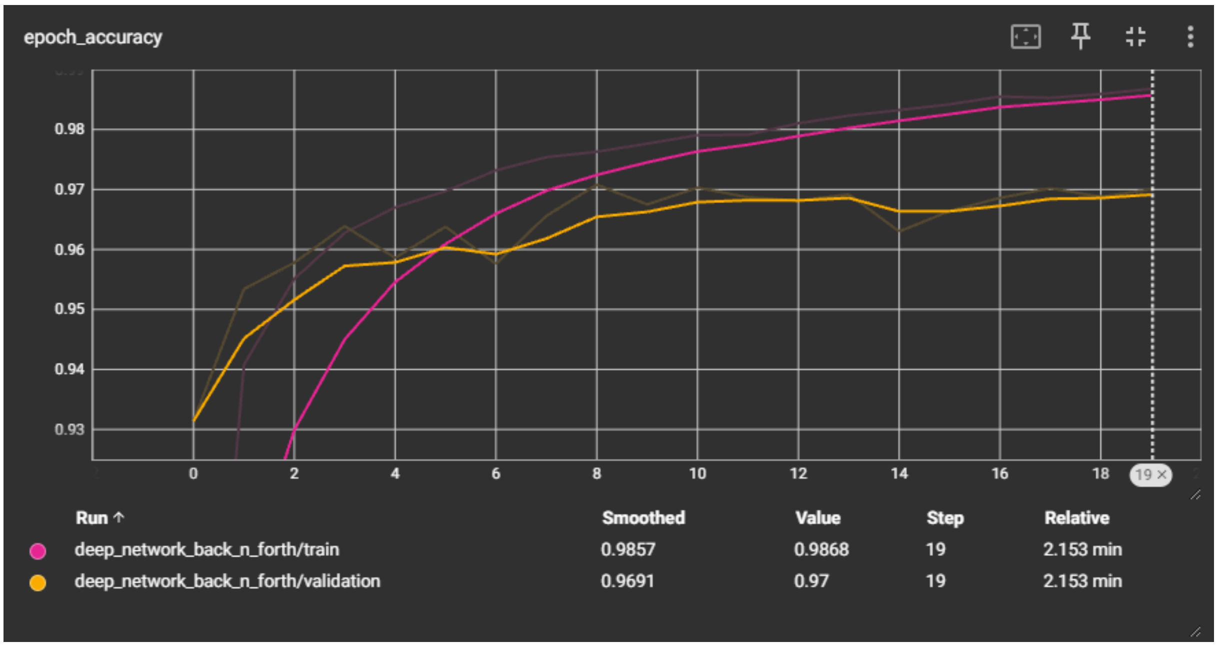
Validation Accuracy: 0.9764

Training Speed: Really, Really Quick (about 2-3 minutes)

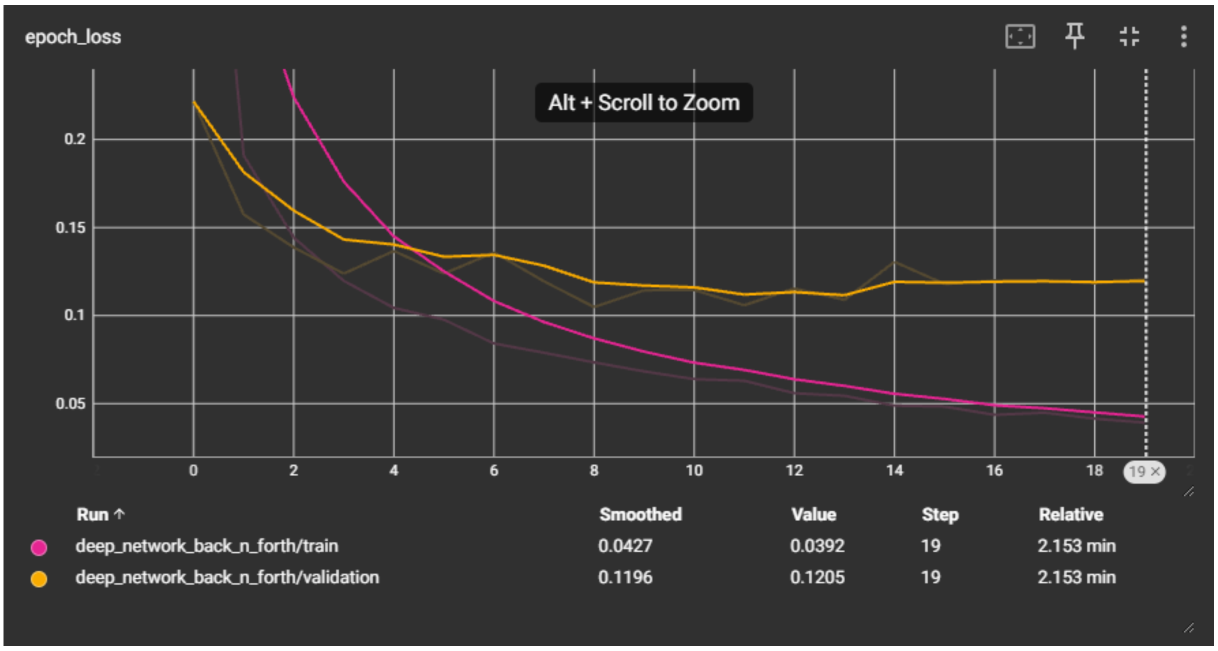
Comment: Pretty good model with mid/high accuracy. Converged very quickly. Epoch loss very small. Quick to train. There are some significant differences between training and testing accuracies.

The idea behind this architecture was to test impact on accuracy the drastic changes between sizes of layers.

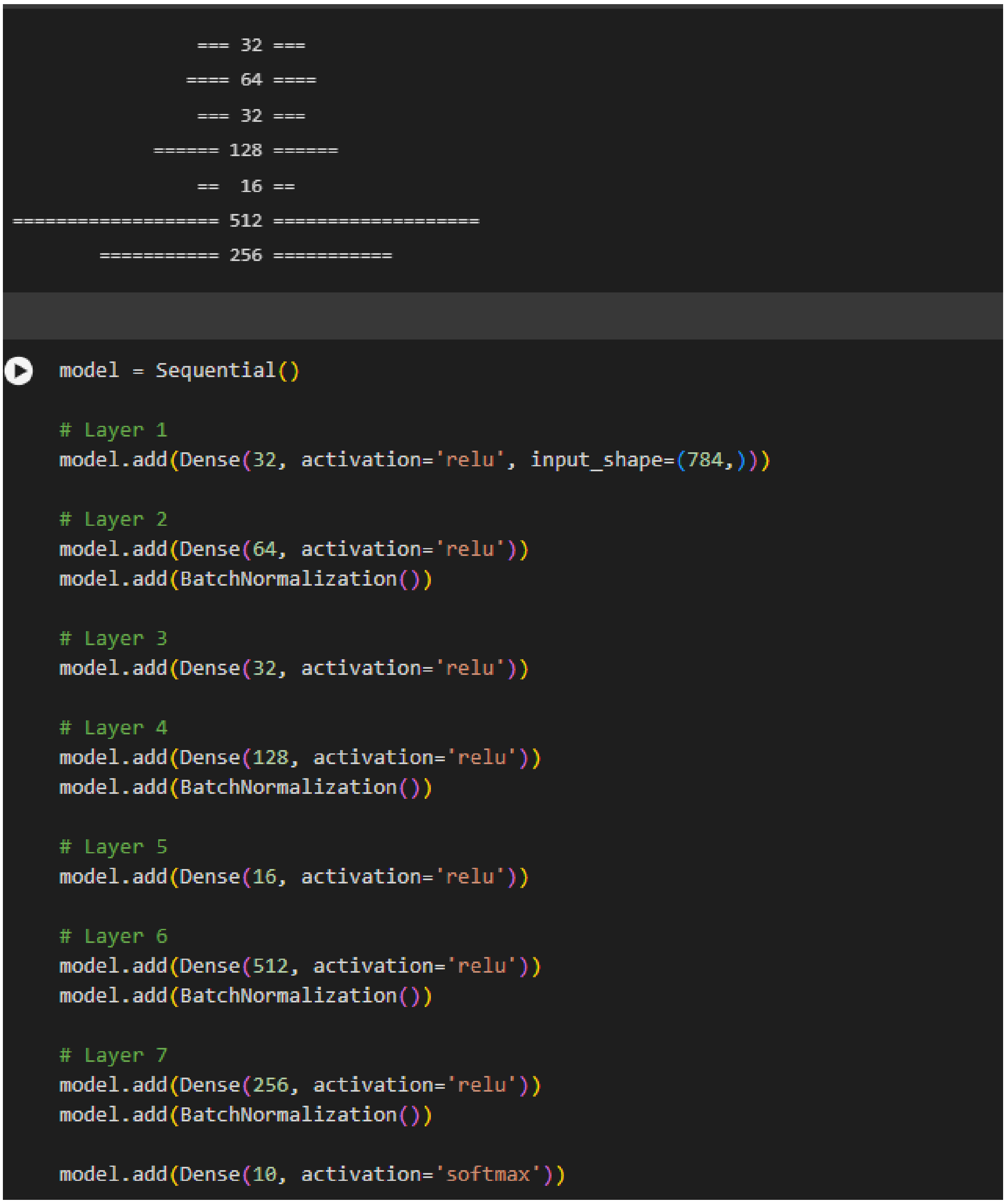
12. Back N Forth



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_back_n_forth/train	0.9857	0.9868	19	2.153 min
●	deep_network_back_n_forth/validation	0.9691	0.97	19	2.153 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_back_n_forth/train	0.0427	0.0392	19	2.153 min
●	deep_network_back_n_forth/validation	0.1196	0.1205	19	2.153 min



```
                 === 32 ===
                ==== 64 ====
                 === 32 ===
             ====== 128 ======
                 ==  16 ==
=================== 512 ===================
        =========== 256 ===========
```

```python
model = Sequential()

# Layer 1
model.add(Dense(32, activation='relu', input_shape=(784,)))

# Layer 2
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())

# Layer 3
model.add(Dense(32, activation='relu'))

# Layer 4
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())

# Layer 5
model.add(Dense(16, activation='relu'))

# Layer 6
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())

# Layer 7
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(10, activation='softmax'))
```

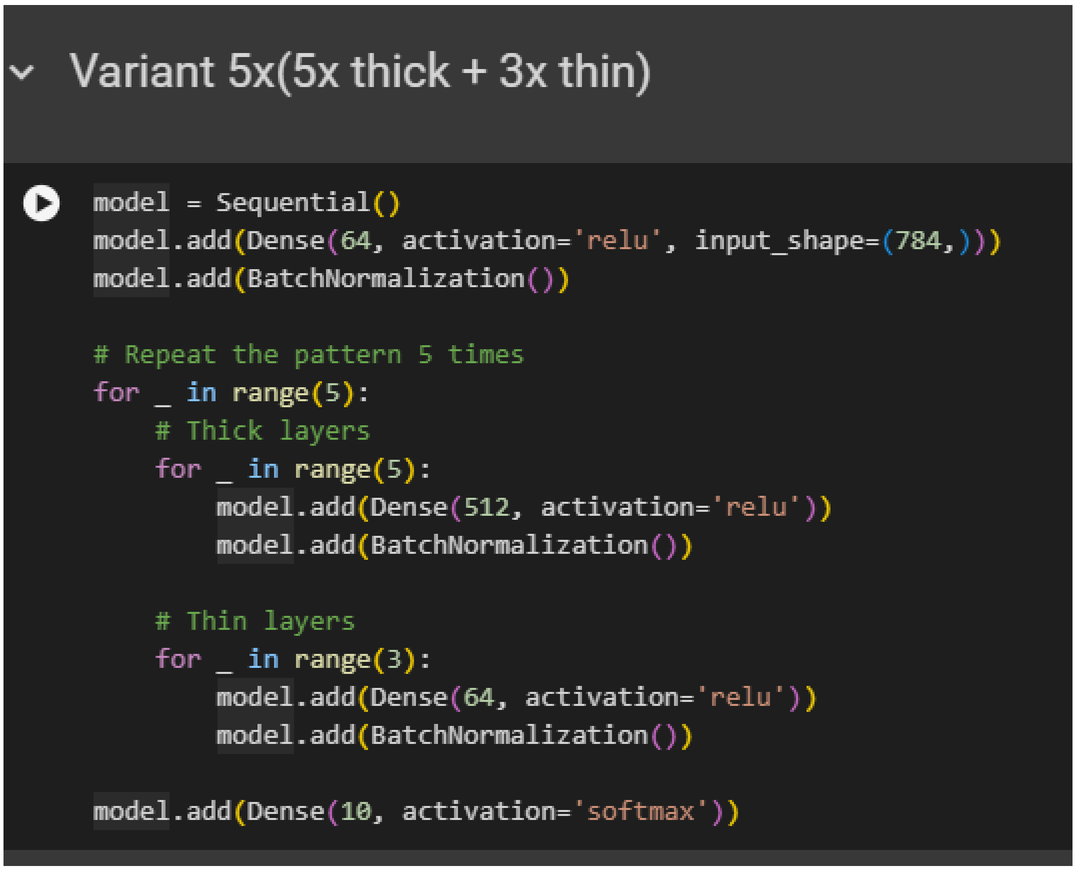
Validation Accuracy: 0.97

Training Speed: Quick (about 2-3 minutes)

Comment:

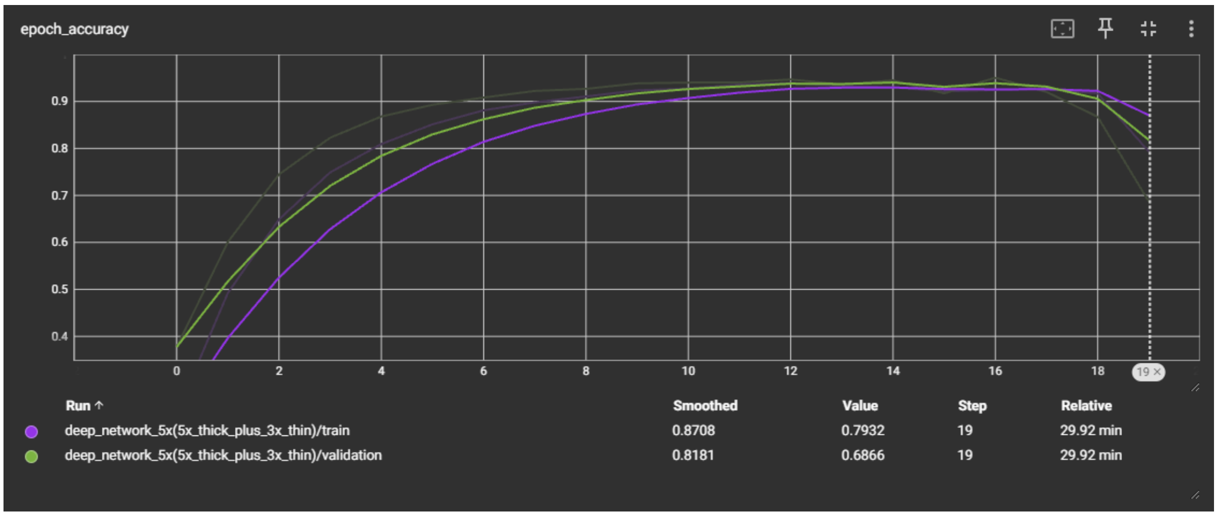
The idea behind this model was to test different layer size while adhiring to the principle of widening and shrinking layers again. This time however introducing 2 wider layers at the end and 1 narrow layer in the middle of the model.
The results are mid/good.

13. 5x (Thick x5 + Thin x3)

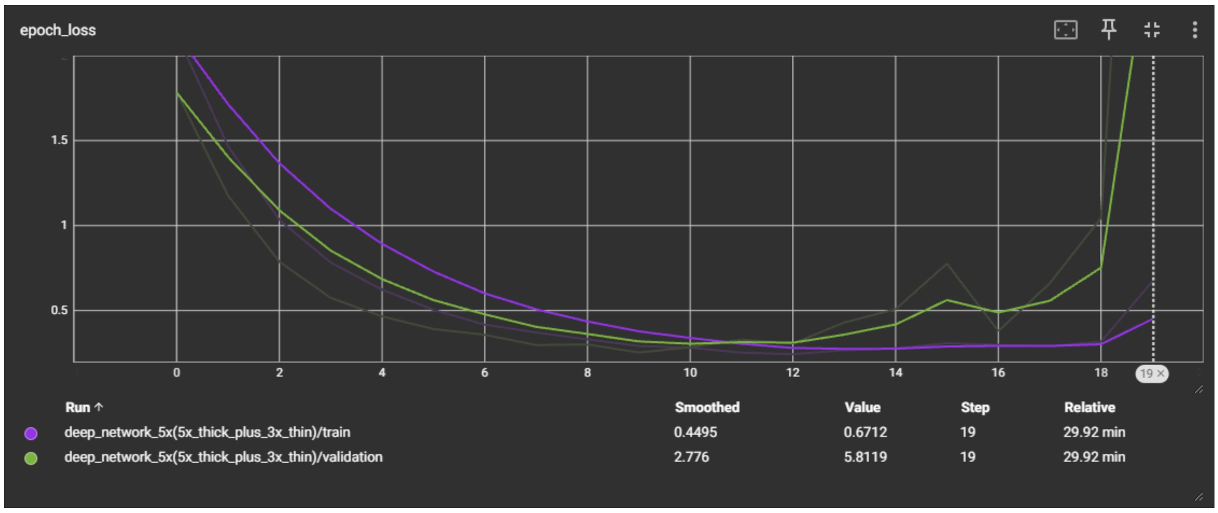


Variant 5x(5x thick + 3x thin)

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

# Repeat the pattern 5 times
for _ in range(5):
    # Thick layers
    for _ in range(5):
        model.add(Dense(512, activation='relu'))
        model.add(BatchNormalization())

    # Thin layers
    for _ in range(3):
        model.add(Dense(64, activation='relu'))
        model.add(BatchNormalization())

model.add(Dense(10, activation='softmax'))
```



Run ↑	Smoothed	Value	Step	Relative
● deep_network_5x(5x_thick_plus_3x_thin)/train	0.8708	0.7932	19	29.92 min
● deep_network_5x(5x_thick_plus_3x_thin)/validation	0.8181	0.6866	19	29.92 min



Run ↑	Smoothed	Value	Step	Relative
● deep_network_5x(5x_thick_plus_3x_thin)/train	0.4495	0.6712	19	29.92 min
● deep_network_5x(5x_thick_plus_3x_thin)/validation	2.776	5.8119	19	29.92 min

Validation Accuracy: 0.68

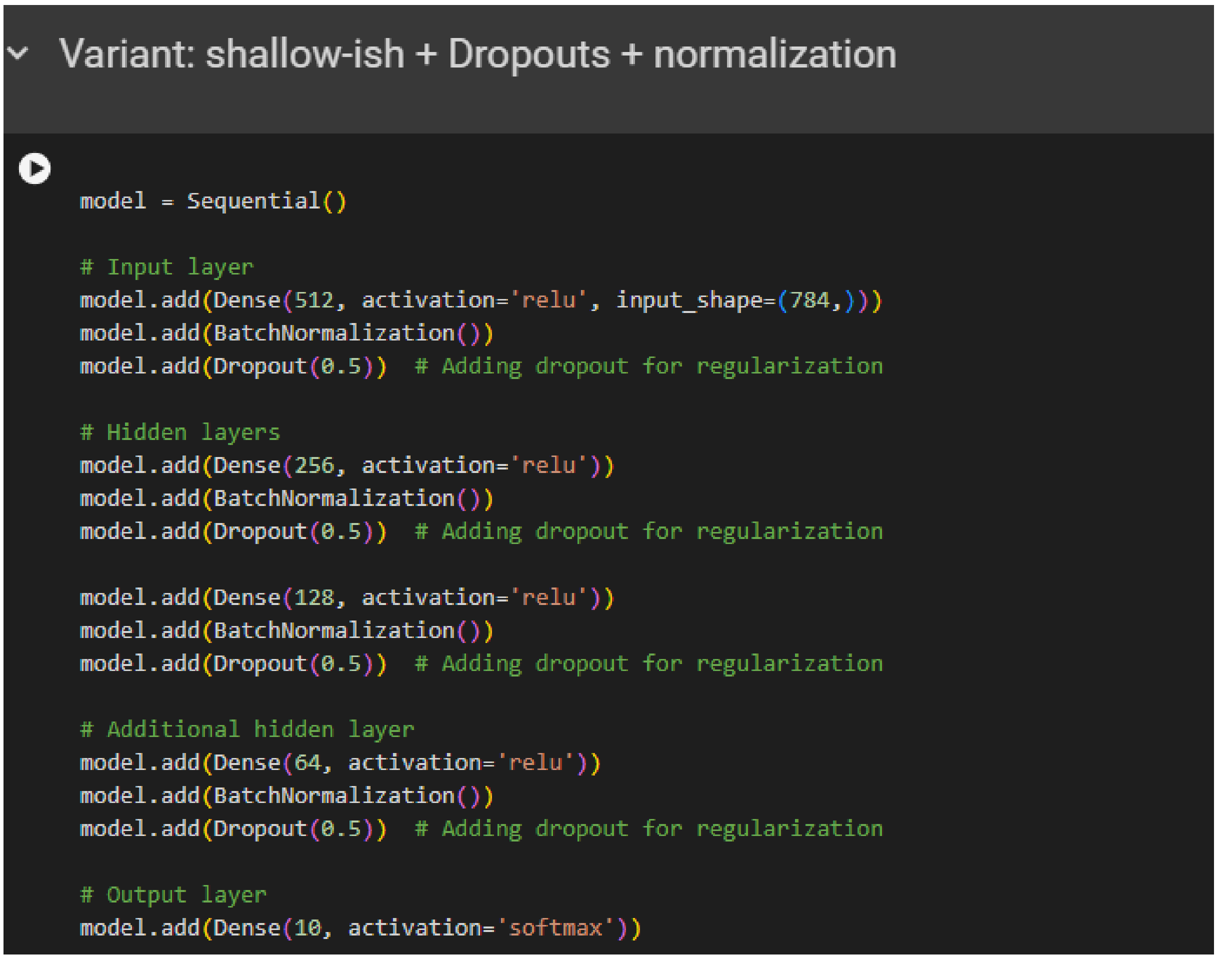
Training Speed: super slow - about 29 minutes

Comment.

The idea behind this architecture was to test impact on accuracy of adding many layers in specific pattern: 5 wider layers and right afterwards 3 smaller. The pattern was repeated 5 times.

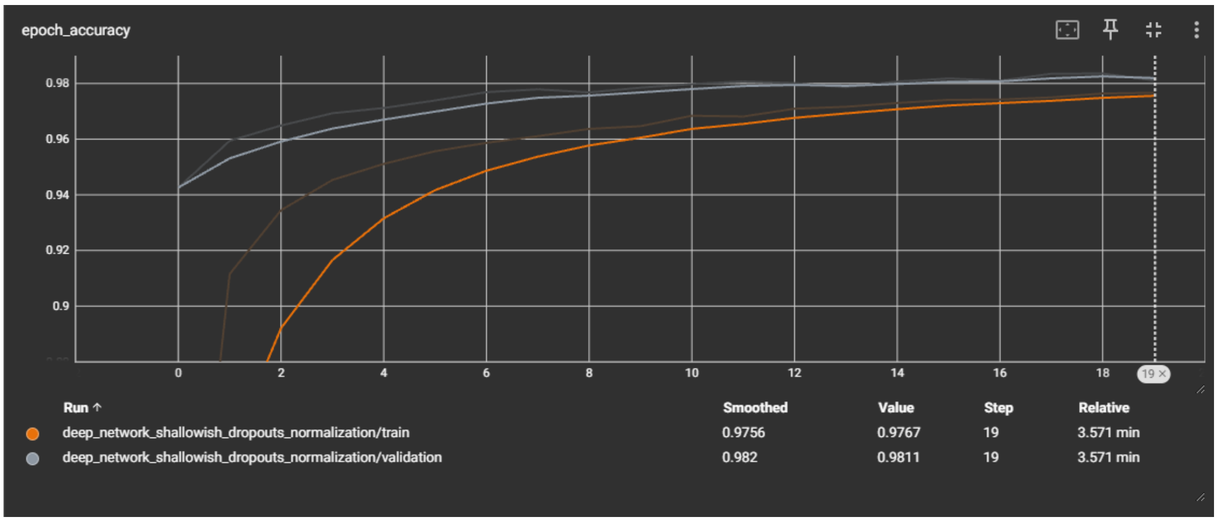
On the contrary to the expectation models heavily underperformed and is absolutely unusable with range of accuracy. Additionally, its training isn't stable since accuracies in each epoch were drastically changing.

14. Shallow-ish but with dropouts and normalization

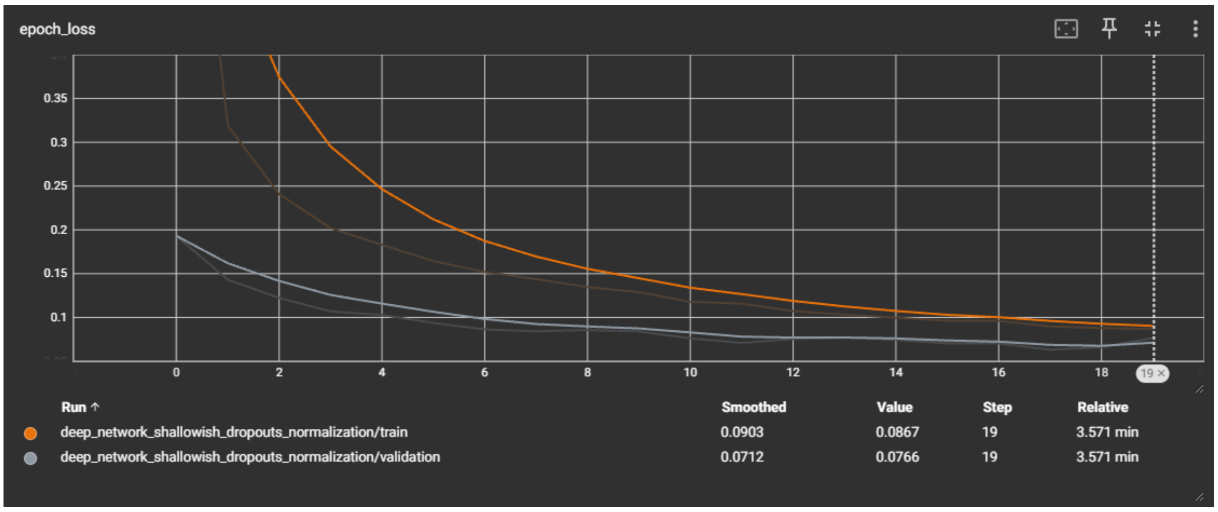


Variant: shallow-ish + Dropouts + normalization

```python
model = Sequential()

# Input layer
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())
model.add(Dropout(0.5))  # Adding dropout for regularization

# Hidden layers
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))  # Adding dropout for regularization

model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))  # Adding dropout for regularization

# Additional hidden layer
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))  # Adding dropout for regularization

# Output layer
model.add(Dense(10, activation='softmax'))
```



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_shallowish_dropouts_normalization/train	0.9756	0.9767	19	3.571 min
●	deep_network_shallowish_dropouts_normalization/validation	0.982	0.9811	19	3.571 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_shallowish_dropouts_normalization/train	0.0903	0.0867	19	3.571 min
●	deep_network_shallowish_dropouts_normalization/validation	0.0712	0.0766	19	3.571 min

Validation Accuracy: Achieved a high validation accuracy of 0.9811, indicating strong performance on unseen data.

Training Speed: Trained very quickly, taking only about 2-3 minutes. This suggests efficient training, which is desirable for practical applications.

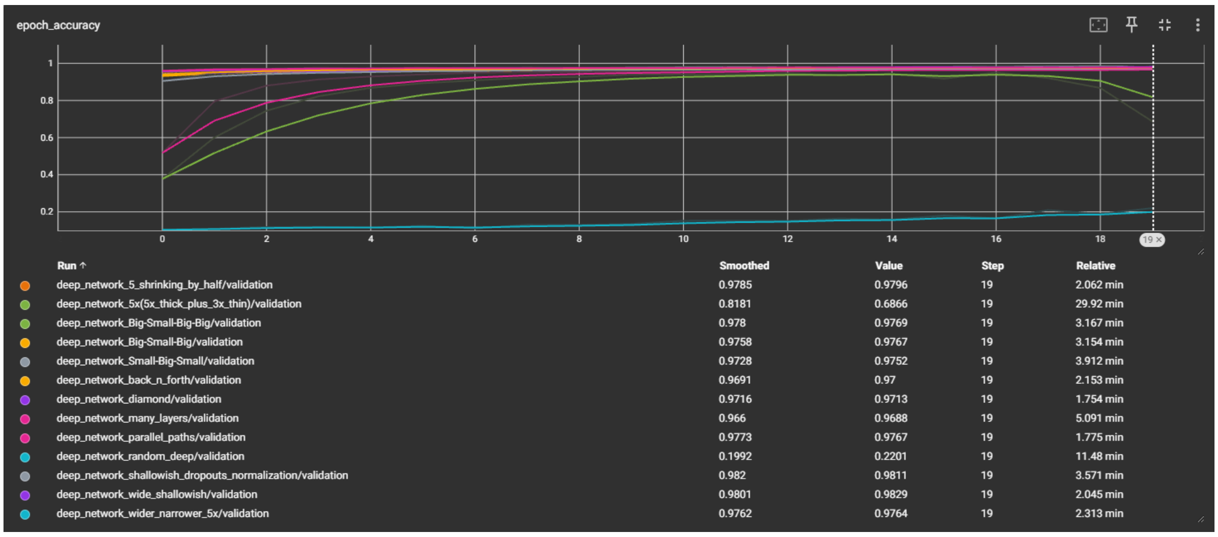
Comment:

One of the best models trained during this session. High accuracy and very quick to train Describes it as one of the best models trained during the session due to its high accuracy and quick training time. Validation and Training accuracies are very close.

The idea behind this architecture was to test impact on accuracy of introducing dropouts. As it turned out its highly beneficial for the model to have some portion of neurons randomly deactivated. For this model Dropout rate was set to half of all neurons.

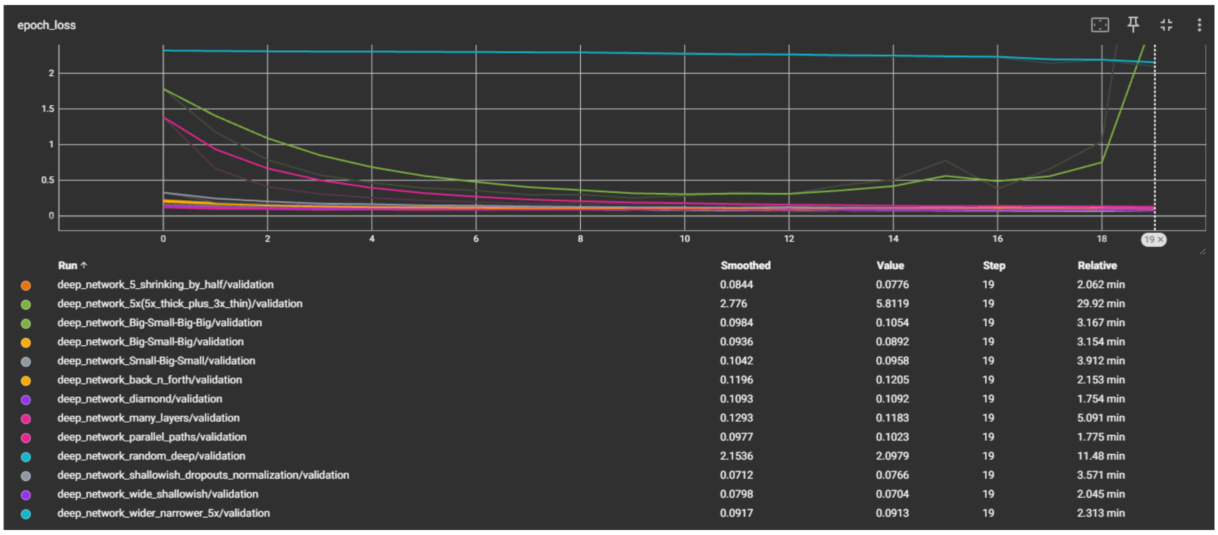
Results visualization

The charts below illustrate the evolution of accuracy and epoch loss for various models across subsequent epochs. Outliers have been identified and are depicted in the charts. The following three charts focus solely on the comparison of the best-performing solutions.



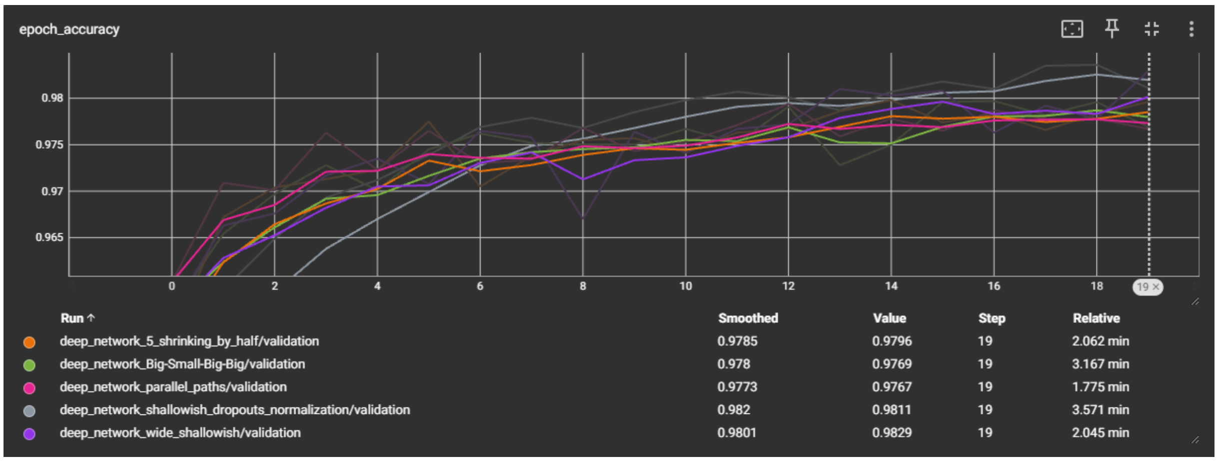
	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_5_shrinking_by_half/validation	0.9785	0.9796	19	2.062 min
●	deep_network_5x(5x_thick_plus_3x_thin)/validation	0.8181	0.6866	19	29.92 min
●	deep_network_Big-Small-Big-Big/validation	0.978	0.9769	19	3.167 min
●	deep_network_Big-Small-Big/validation	0.9758	0.9767	19	3.154 min
●	deep_network_Small-Big-Small/validation	0.9728	0.9752	19	3.912 min
●	deep_network_back_n_forth/validation	0.9691	0.97	19	2.153 min
●	deep_network_diamond/validation	0.9716	0.9713	19	1.754 min
●	deep_network_many_layers/validation	0.966	0.9688	19	5.091 min
●	deep_network_parallel_paths/validation	0.9773	0.9767	19	1.775 min
●	deep_network_random_deep/validation	0.1992	0.2201	19	11.48 min
●	deep_network_shallowish_dropouts_normalization/validation	0.982	0.9811	19	3.571 min
●	deep_network_wide_shallowish/validation	0.9801	0.9829	19	2.045 min
●	deep_network_wider_narrower_5x/validation	0.9762	0.9764	19	2.313 min

Figure 1 Epoch accuracy - all architectures



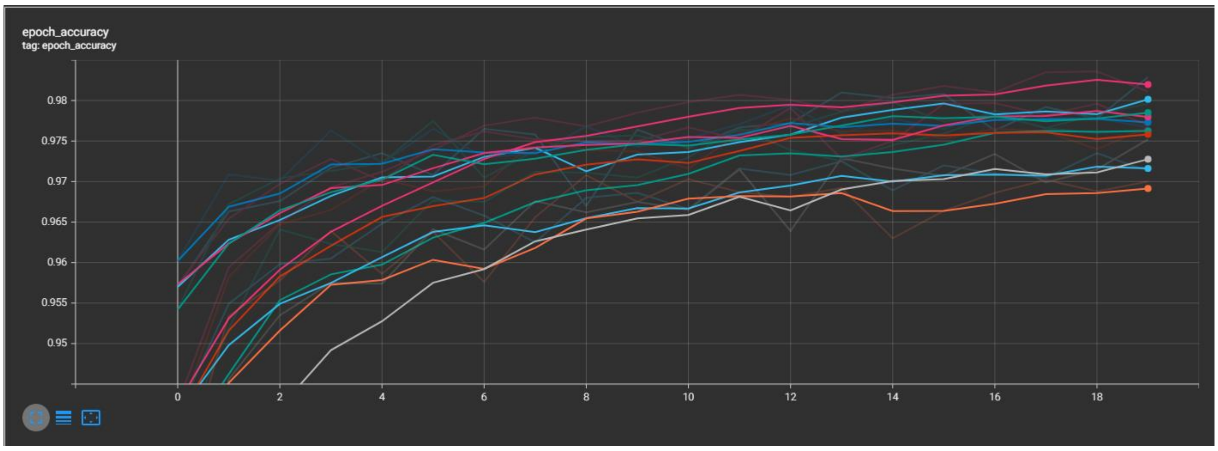
	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_5_shrinking_by_half/validation	0.0844	0.0776	19	2.062 min
●	deep_network_5x(5x_thick_plus_3x_thin)/validation	2.776	5.8119	19	29.92 min
●	deep_network_Big-Small-Big-Big/validation	0.0984	0.1054	19	3.167 min
●	deep_network_Big-Small-Big/validation	0.0936	0.0892	19	3.154 min
●	deep_network_Small-Big-Small/validation	0.1042	0.0958	19	3.912 min
●	deep_network_back_n_forth/validation	0.1196	0.1205	19	2.153 min
●	deep_network_diamond/validation	0.1093	0.1092	19	1.754 min
●	deep_network_many_layers/validation	0.1293	0.1183	19	5.091 min
●	deep_network_parallel_paths/validation	0.0977	0.1023	19	1.775 min
●	deep_network_random_deep/validation	2.1536	2.0979	19	11.48 min
●	deep_network_shallowish_dropouts_normalization/validation	0.0712	0.0766	19	3.571 min
●	deep_network_wide_shallowish/validation	0.0798	0.0704	19	2.045 min
●	deep_network_wider_narrower_5x/validation	0.0917	0.0913	19	2.313 min

Figure 2 Epoch Loss - All Architectures



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_5_shrinking_by_half/validation	0.9785	0.9796	19	2.062 min
●	deep_network_Big-Small-Big-Big/validation	0.978	0.9769	19	3.167 min
●	deep_network_parallel_paths/validation	0.9773	0.9767	19	1.775 min
●	deep_network_shallowish_dropouts_normalization/validation	0.982	0.9811	19	3.571 min
●	deep_network_wide_shallowish/validation	0.9801	0.9829	19	2.045 min

Figure 3 Epoch accuracy : Top 5 archtectures



Summary

In this study, I investigated the impact of model depth and size on the performance of deep learning models. I experimented with various architectures, varying the number of layers and neurons, to understand how these factors affect the model's learning capacity and generalization ability.

My findings suggest that the depth and size of a deep learning model play crucial roles in determining its performance. I observed that deeper models with more layers tend to capture intricate patterns and features in the data, leading to potentially higher accuracy on complex tasks. However, increasing the depth beyond a certain point may result in diminishing returns or even overfitting, especially with limited training data.

On the other hand, the size of the model, determined by the number of neurons or parameters, also influences its performance. Larger models have higher expressive power and can potentially learn more complex relationships in the data. However, they are also more prone to overfitting, especially when the dataset is small or the model lacks regularization techniques.

Importantly, my experiments also highlighted that overcomplicating models meant for simple tasks doesn't always yield better results. In some cases, simpler models with fewer layers and parameters outperform more complex ones, especially when the task is relatively straightforward.

In this case particularly models with maybe 6 layers outperformed much bigger and deeper one.

Additionally, incorporating regularization techniques such as dropout layers proved to be effective in improving model performance. Dropouts help prevent overfitting by randomly deactivating neurons during training, encouraging the model to learn more robust and generalizable representations of the data.

In conclusion, the optimal depth and size of a deep learning model depend on various factors, including the complexity of the task, the size of the dataset, and computational resources. By carefully tuning these parameters and leveraging techniques such as regularization, researchers and practitioners can develop deep learning models that achieve high performance and generalization across different domains and applications.

Sidenotes

Epoch Loss

"Epoch loss" refers to the loss computed over the entire dataset during a single epoch of training in a machine learning model.

During the training process, the dataset is typically divided into batches, and the model's parameters (weights and biases) are updated based on the average loss computed over each batch. After processing all batches in the dataset, the average loss across all batches for that epoch is computed, and this is referred to as the epoch loss.

Epoch loss is a useful metric for monitoring the training progress of a model over multiple epochs. It indicates how well the model is learning from the training data, with lower values indicating better performance. By tracking epoch loss over epochs, you can observe whether the model is converging and improving its performance over time or if there are issues such as overfitting or underfitting.

—

Loss: Loss measures how well the model is performing its task. It's a numerical value that indicates the difference between the predicted output and the actual output.

Epoch: An epoch is one complete pass through the entire dataset during the training of a machine learning model.

Epoch Loss: During each epoch, the model makes predictions on all the training examples, computes the loss for each prediction, and then averages these losses. This average loss across all the examples in the dataset for that epoch is called the "epoch loss."

So, the epoch loss gives you an idea of how well the model is learning from the training data during each epoch of training. Lower epoch loss values indicate that the model is performing better on the training data.